
alibi Documentation

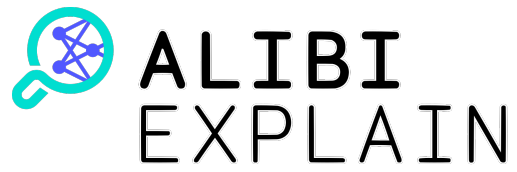
Release 0.9.5dev

Seldon Technologies Ltd

Mar 08, 2024

OVERVIEW

1	Introduction	3
2	Getting Started	33
3	Algorithm overview	39
4	White-box and black-box models	43
5	Saving and loading	47
6	Frequently Asked Questions	49
7	Methods	53
8	Examples	153
9	Methods	501
10	Examples	507
11	Methods	541
12	Examples	547
13	alibi	561
	Python Module Index	781
	Index	783



[Alibi Explain](#) is an open source Python library aimed at machine learning model inspection and interpretation. The focus of the library is to provide high-quality implementations of black-box, white-box, local and global explanation methods for classification and regression models.

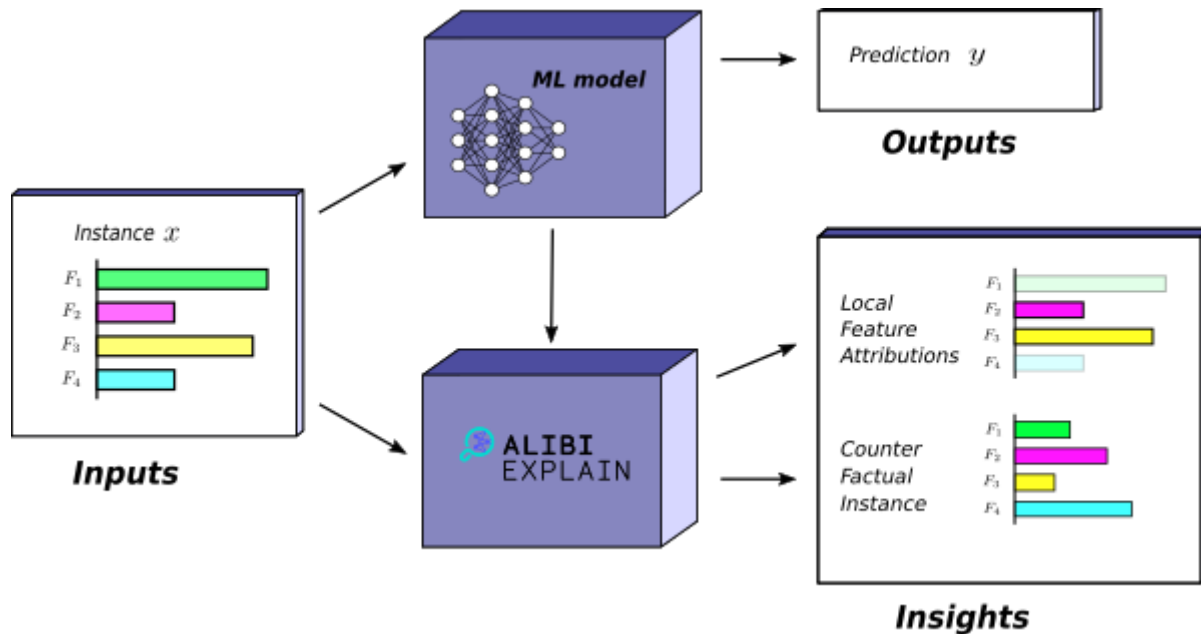
INTRODUCTION

- *What is Explainability?*
 - *Applications*
 - *Black-box vs White-box methods*
 - *Global and Local Insights*
 - *Biases*
- *Types of Insights*
 - *1. Global Feature Attribution*
 - * *Accumulated Local Effects*
 - * *Partial Dependence*
 - * *Partial Dependence Variance*
 - * *Permutation Importance*
 - *2. Local Necessary Features*
 - * *Anchors*
 - * *Contrastive Explanation Method (Pertinent Positives)*
 - *3. Local Feature Attribution*
 - * *Integrated Gradients*
 - * *Kernel SHAP*
 - * *Path-dependent Tree SHAP*
 - * *Interventional Tree SHAP*
 - *4. Counterfactual instances*
 - * *Counterfactual Instances*
 - * *Contrastive Explanation Method (Pertinent Negatives)*
 - * *Counterfactuals Guided by Prototypes*
 - * *Counterfactuals with Reinforcement Learning*
 - * *Counterfactual Example Results*
 - *5. Similarity explanations*

1.1 What is Explainability?

Explainability provides us with algorithms that give insights into trained model predictions. It allows us to answer questions such as:

- How does a prediction **change** dependent on feature inputs?
- What features **are** or **are not** important for a given prediction to hold?
- What set of features would you have to minimally **change** to obtain a **new** prediction of your choosing?
- How does each feature **contribute** to a model's prediction?



Alibi provides a set of **algorithms** or **methods** known as **explainers**. Each explainer provides some kind of insight about a model. The set of insights available given a trained model is dependent on a number of factors. For instance, if the model is a **regression model** it makes sense to ask how the prediction varies for some regressor. Whereas it doesn't make sense to ask what minimal change is required to obtain a new class prediction. In general, given a model the explainers available from **Alibi** are constrained by:

- The **type of data** the model handles. Each insight applies to some or all of the following kinds of data: image, tabular or textual.
- The **task the model** performs. Alibi provides explainers for regression or **classification** models.
- The **type of model** used. Examples of model types include **neural networks** and **random forests**.

1.1.1 Applications

As machine learning methods have become more complex and more mainstream, with many industries now **incorporating AI** in some form or another, the need to understand the decisions made by models is only increasing. Explainability has several applications of importance.

- **Trust:** At a core level, explainability builds **trust** in the machine learning systems we use. It allows us to justify their use in many contexts where an understanding of the basis of the decision is paramount. This is a common issue within machine learning in medicine, where acting on a model prediction may require expensive or risky procedures to be carried out.

- **Testing:** Explainability might be used to [audit financial models](#) that aid decisions about whether to grant customer loans. By computing the attribution of each feature towards the prediction the model makes, organisations can check that they are consistent with human decision-making. Similarly, explainability applied to a model trained on image data can explicitly show the model’s focus when making decisions, aiding [debugging](#). Practitioners must be wary of [misuse](#), however.
- **Functionality:** Insights can be used to augment model functionality. For instance, providing information on top of model predictions such as how to change model inputs to obtain desired outputs.
- **Research:** Explainability allows researchers to understand how and why opaque models make decisions. This can help them understand more broadly the effects of the particular model or training schema they’re using.

1.1.2 Black-box vs White-box methods

Some explainers apply only to specific types of models such as the [Tree SHAP](#) methods which can only be used with [tree-based models](#). This is the case when an explainer uses some aspect of that model’s internal structure. If the model is a neural network then some methods require taking gradients of the model predictions with respect to the inputs. Methods that require access to the model internals are known as **white-box** methods. Other explainers apply to any type of model. They can do so because the underlying method doesn’t make use of the model internals. Instead, they only need to have access to the model outputs given particular inputs. Methods that apply in this general setting are known as **black-box** methods. Typically, white-box methods are faster than black-box methods as they can exploit the model internals. For a more detailed discussion see [white-box and black-box models](#).

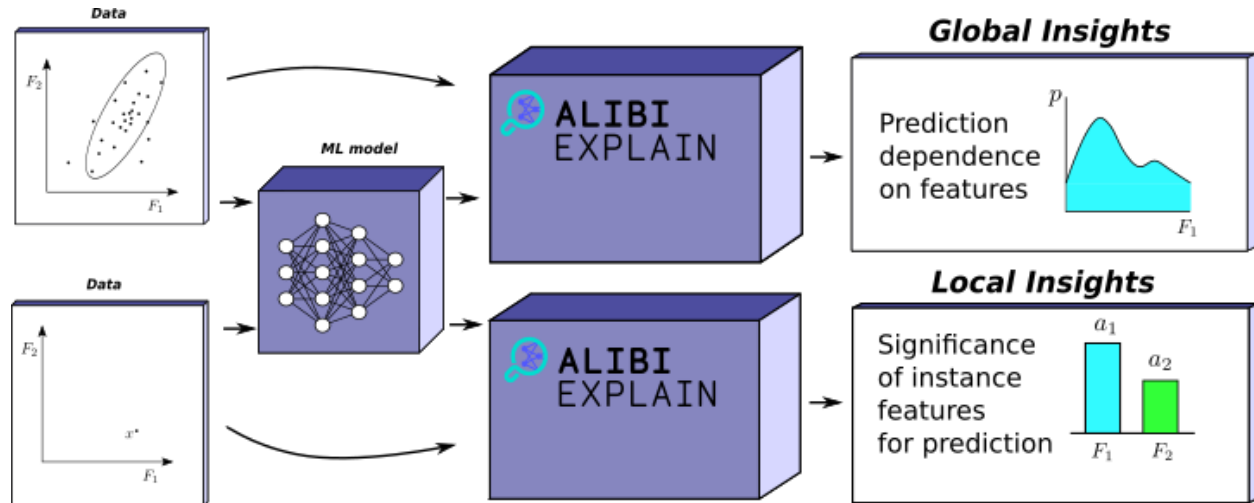
Note 1: Black-box Definition

The use of black-box here varies subtly from the conventional use within machine learning. In most other contexts a model is a black-box if the mechanism by which it makes predictions is too complicated to be interpretable to a human. Here we use black-box to mean that the explainer method doesn’t need access to the model internals to be applied.

1.1.3 Global and Local Insights

Insights can be categorised into two categories — local and global. Intuitively, a local insight says something about a single prediction that a model makes. For example, given an image classified as a cat by a model, a local insight might give the set of features (pixels) that need to stay the same for that image to remain classified as a cat.

On the other hand, global insights refer to the behaviour of the model over a range of inputs. As an example, a plot that shows how a regression prediction varies for a given feature. These insights provide a more general understanding of the relationship between inputs and model predictions.



1.1.4 Biases

The explanations Alibi’s methods provide depend on the model, the data, and — for local methods — the instance of interest. Thus Alibi allows us to obtain insight into the model and, therefore, also the data, albeit indirectly. There are several pitfalls of which the practitioner must be wary.

Often bias exists in the data we feed machine learning models even when we exclude sensitive factors. Ostensibly explainability is a solution to this problem as it allows us to understand the model’s decisions to check if they’re appropriate. However, human bias itself is still an element. Hence, if the model is doing what we expect it to on biased data, we are vulnerable to using explainability to justify relations in the data that may not be accurate. Consider:

“Before launching the model, risk analysts are asked to review the Shapley value explanations to ensure that the model exhibits expected behavior (i.e., the model uses the same features that a human would for the same task).” — [Explainable Machine Learning in Deployment](#)

The critical point here is that the risk analysts in the above scenario must be aware of their own bias and potential bias in the dataset. The Shapley value explanations themselves don’t remove this source of human error; they just make the model less opaque.

Machine learning engineers may also have expectations about how the model should be working. An explanation that doesn’t conform to their expectations may prompt them to erroneously decide that the model is “incorrect”. People usually expect classifiers trained on image datasets to use the same structures humans naturally do when identifying the same classes. However, there is no reason to believe such models should behave the same way we do.

Interpretability of insights can also mislead. Some insights such as [anchors](#) give conditions for a classifiers prediction. Ideally, the set of these conditions would be small. However, when obtaining anchors close to decision boundaries, we may get a complex set of conditions to differentiate that instance from near members of a different class. Because this is harder to understand, one might write the model off as incorrect, while in reality, the model performs as desired.

1.2 Types of Insights

Alibi provides several local and global insights with which to explore and understand models. The following gives the practitioner an understanding of which explainers are suitable in which situations.

Explainer	Scope	Model types	Task types	Data types	Use	Resources
<i>Accumulated Local Effects</i>	Global	Black-box	Classification, Regression	Tabular (numerical)	How does model prediction vary with respect to features of interest?	docs , paper
<i>Partial Dependence</i>	Global	Black-box, White-box (<i>scikit-learn</i>)	Classification, Regression	Tabular (numerical, categorical)	How does model prediction vary with respect to features of interest?	docs , paper
<i>Partial Dependence Variance</i>	Global	Black-box, White-box (<i>scikit-learn</i>)	Classification, Regression	Tabular (numerical, categorical)	Which are the most important features globally? How much do features interact globally?	docs , paper
<i>Permutation importance</i>	Global	Black-box	Classification, Regression	Tabular (numerical, categorical)	Which are the most important features globally?	docs , paper
<i>Anchors</i>	Local	Black-box	Classification	Tabular (numerical, categorical), Text and Image	Which set of features of a given instance are sufficient to ensure the prediction stays the same?	docs , paper
<i>Pertinent Positives</i>	Local	Black-box, White-box (<i>TensorFlow</i>)	Classification	Tabular (numerical), Image	“”	docs , paper
<i>Integrated Gradients</i>	Local	White-box (<i>TensorFlow</i>)	Classification, Regression	Tabular (numerical, categorical), Text and Image	What does each feature contribute to the model prediction?	docs , paper
<i>Kernel SHAP</i>	Local	Black-box	Classification, Regression	Tabular (numerical, categorical)	“”	docs , paper
<i>Tree SHAP (path-dependent)</i>	Local	White-box (<i>XGBoost</i> , <i>LightGBM</i> , <i>CatBoost</i> , <i>scikit-learn</i> and <i>pyspark tree models</i>)	Classification, Regression	Tabular (numerical, categorical)	“”	docs , paper
<i>Tree SHAP (interventional)</i>	Local	White-box (<i>XGBoost</i> , <i>LightGBM</i> , <i>CatBoost</i> , <i>scikit-learn</i> and <i>pyspark tree models</i>)	Classification, Regression	Tabular (numerical, categorical)	“”	docs , paper
<i>Counterfactual</i>	Local	Black-box (<i>differen-</i>	Clas-	Tabular (nu-	What minimal change to fea-	docs ,

1.2.1 1. Global Feature Attribution

Global Feature Attribution methods aim to show the dependency of model output on a subset of the input features. They provide global insight describing the model's behaviour over the input space. For instance, Accumulated Local Effects plots obtain graphs that directly visualize the relationship between feature and prediction over a specific set of samples.

Suppose a trained regression model that predicts the number of bikes rented on a given day depending on the temperature, humidity, and wind speed. A global feature attribution plot for the temperature feature might be a line graph plotted against the number of bikes rented. One would anticipate an increase in rentals until a specific temperature and then a decrease after it gets too hot.

Accumulated Local Effects

Explainer	Scop	Model types	Task types	Data types	Use	Re-sources
<i>Accumulated Local Effects</i>	Global	Black-box	Classification, Regression	Tabular (numerical)	How does model prediction vary with respect to features of interest?	docs , paper

Alibi provides *accumulated local effects (ALE)* plots because they give the most accurate insight. Alternatives include Partial Dependence Plots (PDP), of which ALE is a natural extension. Suppose we have a model f and features $X = \{x_1, \dots, x_n\}$. Given a subset of the features X_S , we denote $X_C = X \setminus X_S$. X_S is usually chosen to be of size at most 2 in order to make the generated plots easy to visualize. PDP works by marginalizing the model's output over the features we are not interested in, X_C . The process of factoring out the X_C set causes the introduction of artificial data, which can lead to errors. ALE plots solve this by using the conditional probability distribution instead of the marginal distribution and removing any incorrect output dependencies due to correlated input variables by accumulating local differences in the model output instead of averaging them. See the [following](#) for a more expansive explanation.

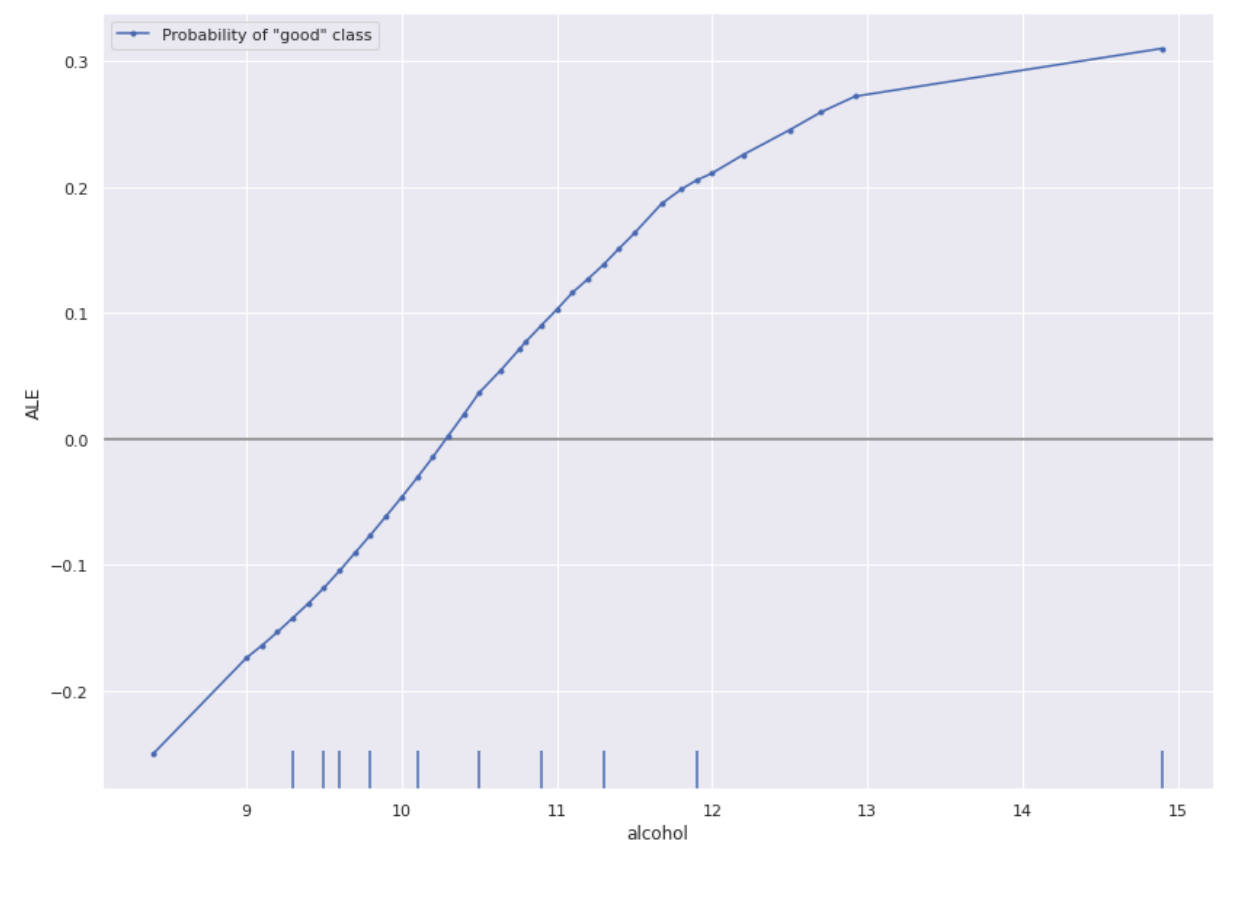
We illustrate the use of ALE on the [wine-quality dataset](#) which is a tabular numeric dataset with wine quality as the target variable. Because we want a classification task we split the data into good and bad classes using 5 as the threshold. We can compute the ALE with Alibi (see [notebook](#)) by simply using:

```
from alibi.explainers import ALE, plot_ale

# Model is a binary classifier so we only take the first model output corresponding to
# "good" class probability.
predict_fn = lambda x: model(scaler.transform(x)).numpy()[0]
ale = ALE(predict_fn, feature_names=features)
exp = ale.explain(X_train)

# Plot the explanation for the "Alcohol feature"
plot_ale(exp, features=['alcohol'], line_kw={'label': 'Probability of "good" class'})
```

Hence, we see the model predicts higher alcohol content wines as being better:



Note 2: Categorical Variables and ALE

Note that while ALE is well-defined on numerical tabular data, it isn't on categorical data. This is because it's unclear what the difference between two categorical values should be. Note that if the dataset has a mix of categorical and numerical features, we can always compute the ALE of the numerical ones.

Pros	Cons
ALE plots are easy to visualize and understand intuitively	Harder to explain the underlying motivation behind the method than PDP plots or M plots
Very general as it is a black-box algorithm	Requires access to the training dataset
Doesn't struggle with dependencies in the underlying features, unlike PD plots	ALE of categorical variables is not well-defined
ALE plots are fast	

Partial Dependence

Ex- plainer	Scop	Model types	Task types	Data types	Use	Re- sources
<i>Partial Dependence</i>	Glob	Black-box, White-box (scikit-learn)	Classi- fication, Regression	Tabular (nu- merical, categorical)	How does model prediction vary with respect to features of interest?	docs , paper

Alibi provides *partial dependence (PD)* plots as an alternative to ALE. Following the same notation as above, we remind the reader that the PD is marginalizing the model's output over the features we are not interested in, X_C . This approach has a direct extension for categorical features, something that ALE struggle with. Although, the practitioner should be aware of the main limitation of PD, which is the assumption of feature independence. The process of marginalizing out the set X_C under the assumption of feature independence might thus include in the computation predictions for data instances belonging to low probability regions of the features distribution.

Pros	Cons
PD plots are easy to visualize and understand intuitively (easier than ALE)	Struggle with dependencies in the underlying features. In the uncorrelated case the interpretation might be unclear. Heterogeneous effects might be hidden (ICE to the rescue)
Very general as it is a black-box algorithm	
PD plots are in general fast. Even faster implementation for scikit-learn tree based models	
PD plots have causal interpretation. The relationship is causal for the model, but not necessarily for the real world	
Natural extension to categorical features	

Partial Dependence Variance

Explainer	Scop	Model types	Task types	Data types	Use	Re- sources
<i>Partial Dependence Variance</i>	Glob	Black-box, White-box (scikit-learn)	Classi- fication, Regres- sion	Tabular (nu- merical, cat- egorical)	What are the most important features globally? How much do features interact globally?	docs , pa- per

Alibi provides *partial dependence variance* as a way to measure globally the feature importance and the strength of the feature interactions between pairs of features. Since the method is based on the partial dependence, the practitioner should be aware that the method inherits its main limitations (see discussion above).

Pros	Cons
Intuitive motivation for the computation of the feature importance	The feature importance captures only the main effect and ignores possible feature interaction
Very general as it is a black-box algorithm	Can fail to detect feature interaction even though those exist
Fast computation in general. Even faster implementation for scikit-learn tree-based models	
Offers standardized procedure to quantify the feature importance (i.e., contrasts with internal feature importance for some tree-based model)	
Offers support for both numerical and categorical features	
Can quantify the strength of potential interaction effects	

Permutation Importance

Explainer	Scope	Model types	Task types	Data types	Use	Resources
<i>Permutation Importance</i>	Global	Black-box	Classification, Regression	Tabular (numerical, categorical)	Which are the most important features globally?	docs , paper

Alibi provides *permutation importance* as a way to measure globally the feature importance. The computation of the feature importance is based on the degree of model performance degradation when the feature values within a feature column are permuted. One important behavior that a practitioner should be aware of is that the importance of correlated features can be split between them.

Pros	Cons
A nice and simple interpretation - the feature importance is the increase/decrease in the model loss/score when a feature is noise.	Need the ground truth labels
Very general as it is a black-box algorithm	Can be biased towards unrealistic data instances
The feature importance takes into account all the feature interactions	The importance metric is related to the loss/score function
Does not require retraining the model	

1.2.2 2. Local Necessary Features

Local necessary features tell us what features need to stay the same for a specific instance in order for the model to give the same classification. In the case of a trained image classification model, local necessary features for a given instance would be a minimal subset of the image that the model uses to make its decision. Alibi provides two explainers for computing local necessary features: *anchors* and *pertinent positives*.

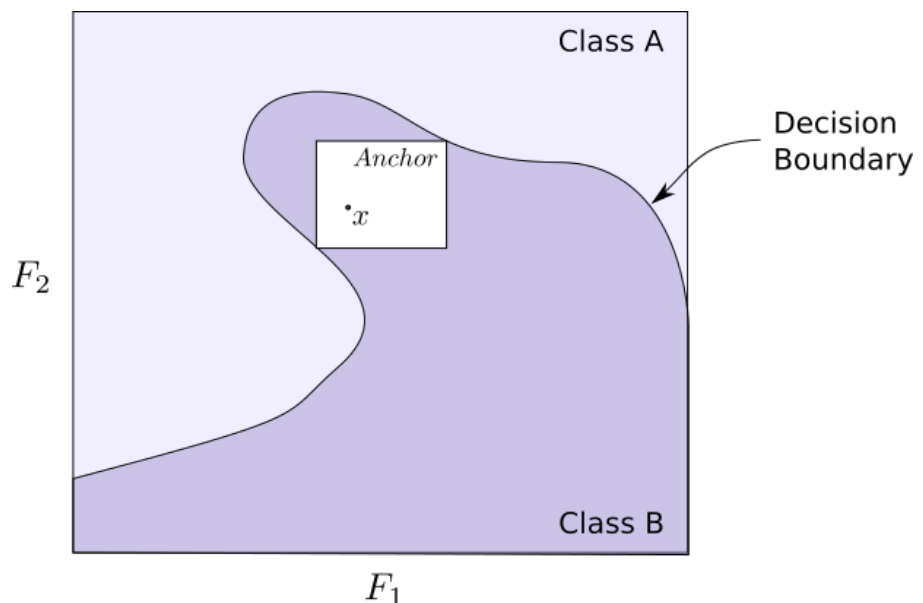
Anchors

Ex- plaine	Scoꝛ	Model types	Task types	Data types	Use	Re- sources
<i>An- chors</i>	Lo- cal	Black- box	Clas- sifica- tion	Tabular (numerical, categorical), Text and Image	Which set of features of a given instance are sufficient to ensure the prediction stays the same?	docs , paper

Anchors are introduced in [Anchors: High-Precision Model-Agnostic Explanations](#). More detailed documentation can be found [here](#).

Let A be a rule (set of predicates) acting on input instances, such that $A(x)$ returns 1 if all its feature predicates are true. Consider the [wine quality dataset](#) adjusted by partitioning the data into good and bad wine based on a quality threshold of 0.5:

fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	class
7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	bad
7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	bad
7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	bad
11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	good
7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	bad
...



An example of a predicate for this dataset would be a rule of the form: `'alcohol > 11.00'`. Note that the more predicates we add to an anchor, the fewer instances it applies to, as by doing so, we filter out more instances of the data.

Anchors are sets of predicates associated to a specific instance x such that x is in the anchor ($A(x) = 1$) and any other point in the anchor has the same classification as x (z such that $A(z) = 1 \implies f(z) = f(x)$ where f is the model). We're interested in finding the Anchor that contains both the most instances and also x .

To construct an anchor using Alibi for tabular data such as the wine quality dataset (see [notebook](#)), we use:

```
from alibi.explainers import AnchorTabular

predict_fn = lambda x: model.predict(scaler.transform(x))
explainer = AnchorTabular(predict_fn, features)
explainer.fit(X_train)

# x is the instance to explain
result = explainer.explain(x)

print('Anchor = ', result.data['anchor'])
print('Coverage = ', result.data['coverage'])
```

where x is an instance of the dataset classified as good.

```
Mean test accuracy 95.00%
Anchor = ['sulphates <= 0.55', 'volatile acidity > 0.52', 'alcohol <= 11.00', 'pH > 3.40
↪']
Coverage = 0.0316930775646372
```

Note: Alibi also gives an idea of the size (coverage) of the Anchor which is the proportion of the input space the anchor applies to.

To find anchors Alibi sequentially builds them by generating a set of candidates from an initial anchor candidate, picking the best candidate of that set and then using that to generate the next set of candidates and repeating. Candidates are favoured on the basis of the number of instances they contain that are in the same class as x under f . The proportion of instances the anchor contains that are classified the same as x is known as the *precision* of the anchor. We repeat the above process until we obtain a candidate anchor with satisfactory precision. If there are multiple such anchors we choose the one that contains the most instances (as measured by *coverage*).

To compute which of two anchors is better, Alibi obtains an estimate by sampling from $\mathcal{D}(z|A)$ where \mathcal{D} is the data distribution. The sampling process is dependent on the type of data. For tabular data, this process is easy; we can fix the values in the Anchor and replace the rest with values from points sampled from the dataset.

In the case of textual data, anchors are sets of words that the sentence must include to be **in the** anchor. To sample from $\mathcal{D}(z|A)$, we need to find realistic sentences that include those words. To help do this Alibi provides support for three [transformer](#) based language models: `DistilbertBaseUncased`, `BertBaseUncased`, and `RobertaBase`.

Image data being high-dimensional means we first need to reduce it to a lower dimension. We can do this using image segmentation algorithms (Alibi supports [felzenszwalb](#), [slic](#) and [quickshift](#)) to find super-pixels. The user can also use their own custom defined segmentation function. We then create the anchors from these super-pixels. To sample from $\mathcal{D}(z|A)$ we replace those super-pixels that aren't in A with something else. Alibi supports superimposing over the absent super-pixels with an image sampled from the dataset or taking the average value of the super-pixel.

The fact that the method requires perturbing and comparing anchors at each stage leads to some limitations. For instance, the more features, the more candidate anchors you can obtain at each process stage. The algorithm uses a [beam search](#) among the candidate anchors and solves for the best B anchors at each stage in the process by framing the problem as a [multi-armed bandit](#). The runtime complexity is $\mathcal{O}(B \cdot p^2 + p^2 \cdot \mathcal{O}_{MAB[B \cdot p, B]})$ where p is the number of features and $\mathcal{O}_{MAB[B \cdot p, B]}$ is the runtime for the multi-armed bandit (see [Molnar](#) for more details).

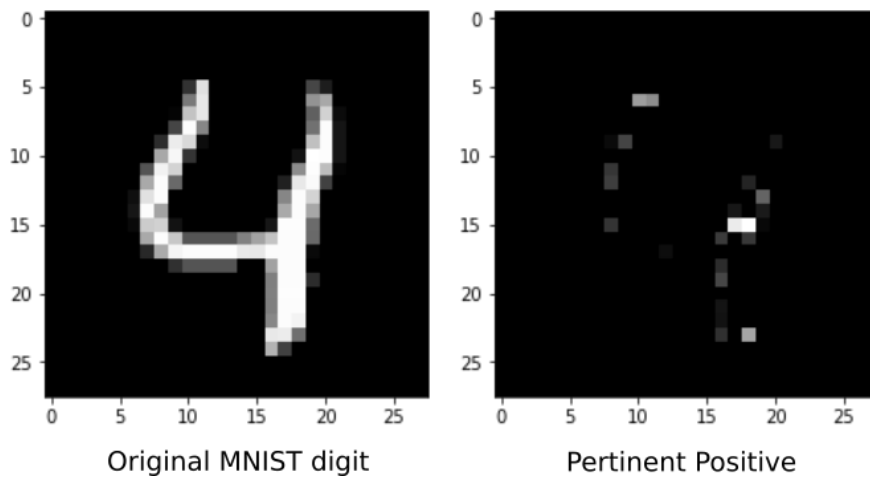
Similarly, comparing anchors that are close to decision boundaries can require many samples to obtain a clear winner between the two. Also, note that anchors close to decision boundaries are likely to have many predicates to ensure the required predictive property. This makes them less interpretable.

Pros	Cons
Easy to explain as rules are simple to interpret	Time complexity scales as a function of features
Is a black-box method as we need to predict the value of an instance and don't need to access model internals	Requires a large number of samples to distinguish anchors close to decision boundaries
The coverage of an anchor gives a level of global insight as well	Anchors close to decision boundaries are less likely to be interpretable
	High dimensional feature spaces such as images need to be reduced to improve the runtime complexity
	Practitioners may need domain-specific knowledge to correctly sample from the conditional probability

Contrastive Explanation Method (Pertinent Positives)

Ex-plainer	Scope	Model types	Task types	Data types	Use	Re-sources
<i>Pertinent Positives</i>	Local	Black-box, White-box (<i>TensorFlow</i>)	Classification	Tabular (numerical), Image	Which set of features of a given instance is sufficient to ensure the prediction stays the same	<i>docs, paper</i>

Introduced by [Amit Dhurandhar, et al](#), a Pertinent Positive is the subset of features of an instance that still obtains the same classification as that instance. These differ from *anchors* primarily in the fact that they aren't constructed to maximize coverage. The method to create them is also substantially different. The rough idea is to define an **absence of a feature** and then perturb the instance to take away as much information as possible while still retaining the original classification. Note that these are a subset of the *CEM* method which is also used to construct pertinent negatives/counterfactuals.



Given an instance x we use gradient descent to find a δ that minimizes the following loss:

$$L = c \cdot L_{pred}(\delta) + \beta L_1(\delta, x) + L_2^2(\delta, x) + \gamma \|\delta - AE(\delta)\|_2^2$$

AE is an *autoencoder* generated from the training data. If δ strays from the original data distribution, the autoencoder loss will increase as it will no longer reconstruct δ well. Thus, we ensure that δ remains close to the original dataset distribution.

Note that δ is constrained to only “take away” features from the instance x . There is a slightly subtle point here: removing features from an instance requires correctly defining non-informative feature values. For the [MNIST digits](#), it’s reasonable to assume that the black background behind each digit represents an absence of information. In general, having to choose a non-informative value for each feature is non-trivial and domain knowledge is required. This is the reverse to the [contrastive explanation method \(pertinent-negatives\)](#) method introduced in the section on counterfactual instances.

Note that we need to compute the loss gradient through the model. If we have access to the internals, we can do this directly. Otherwise, we need to use numerical differentiation at a high computational cost due to the extra model calls we need to make. This does however mean we can use this method for a wide range of black-box models but not all. We require the model to be differentiable which isn’t always true. For instance tree-based models have piece-wise constant output.

Pros	Cons
Can be used with both white-box (Tensor-Flow) and some black-box models	Finding non-informative feature values to take away from an instance is often not trivial, and domain knowledge is essential The autoencoder loss requires access to the original dataset
	Need to tune hyperparameters β and γ
	The insight doesn’t tell us anything about the coverage of the pertinent positive
	Slow for black-box models due to having to numerically evaluate gradients
	Only works for differentiable black-box models

1.2.3 3. Local Feature Attribution

Local feature attribution (LFA) asks how each feature in a given instance contributes to its prediction. In the case of an image, this would highlight those pixels that are most responsible for the model prediction. Note that this differs subtly from Local Necessary Features which find the *minimal subset* of features required to keep the same prediction. Local feature attribution instead assigns a score to each feature.

A good example use of local feature attribution is to detect that an image classifier is focusing on the correct features of an image to infer the class. In their paper “[Why Should I Trust You?: Explaining the Predictions of Any Classifier](#)”, Marco Tulio Ribeiro et al. train a logistic regression classifier on a small dataset of images of wolves and huskies. The data set has been handpicked so that only the pictures of wolves have snowy backdrops while the huskies don’t. LFA methods reveal that the resulting misclassification of huskies in snow as wolves results from the network incorrectly focusing on those images snowy backdrops.

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$. f might be a regression model, a single component of a multi-output regression or a probability of a class in a classification model. If $x = (x_1, \dots, x_n) \in \mathbb{R}^n$ then an attribution of the prediction at input x is a vector $a = (a_1, \dots, a_n) \in \mathbb{R}^n$ where a_i is the contribution of x_i to the prediction $f(x)$.

Alibi exposes four explainers to compute LFAs: [Integrated Gradients](#), [Kernel SHAP](#), [Path-dependent Tree SHAP](#) and [Interventional Tree SHAP](#). The last three of these are implemented in the [SHAP library](#) and Alibi acts as a wrapper. Interventional and path-dependent tree SHAP are white-box methods that apply to tree based models.

For attribution methods to be relevant, we expect the attributes to behave consistently in certain situations. Hence, they should satisfy the following properties.

- **Efficiency/Completeness:** The sum of attributions should equal the difference between the prediction and the baseline



Fig. 1: Figure 11 from “Why Should I Trust You?: Explaining the Predictions of Any Classifier.

- **Symmetry:** Variables that have identical effects on the model should have equal attribution
- **Dummy/Sensitivity:** Variables that don’t change the model output should have attribution zero
- **Additivity/Linearity:** The attribution of a feature for a linear combination of two models should equal the linear combination of attributions of that feature for each of those models

Not all LFA methods satisfy these methods (**LIME** for example) but the ones provided by Alibi (*Integrated Gradients*, *Kernel SHAP*, *Path-dependent* and *Interventional Tree SHAP*) do.

Integrated Gradients

Ex- plainer	Scor	Model types	Task types	Data types	Use	Re- sources
<i>Inte- grated Gradients</i>	Lo- cal	White-box (<i>Tensor- Flow</i>)	Classi- fication, Regression	Tabular (numerical, categorical), Text and Image	What does each feature con- tribute to the model predic- tion?	<i>docs</i> , <i>paper</i>

The *Integrated Gradients* (IG) method computes the attribution of each feature by integrating the model partial derivatives along a path from a baseline point to the instance. This accumulates the changes in the prediction that occur due to the changing feature values. These accumulated values represent how each feature contributes to the prediction for the instance of interest. A more detailed explanation of the method can be found in the method specific *docs*.

We need to choose a baseline which should capture a blank state in which the model makes essentially no prediction or assigns the probability of each class equally. This is dependent on domain knowledge of the dataset. In the case of MNIST for instance a common choice is an image set to black. For numerical tabular data we can set the baseline as the average of each feature.

Note 3: Choice of Baseline

The main difficulty with this method is that as IG is very [dependent on the baseline](#), it's essential to make sure you choose it well. Choosing a black image baseline for a classifier trained to distinguish between photos taken at day or night may not be the best choice.

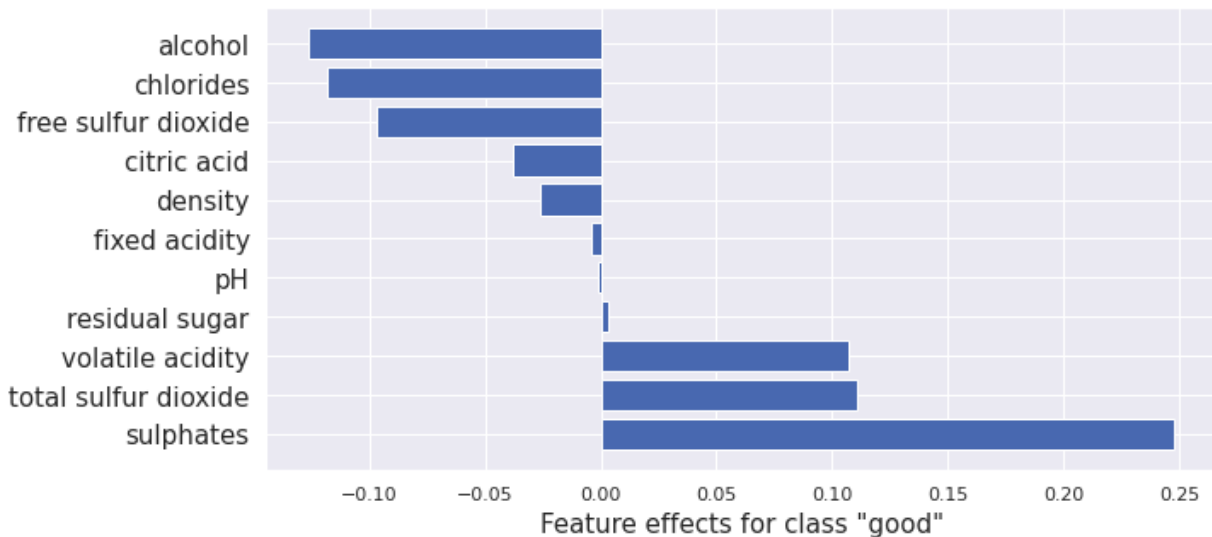
Note that IG is a white-box method that requires access to the model internals in order to compute the partial derivatives. Alibi provides support for TensorFlow models. For example given a TensorFlow classifier trained on the wine quality dataset we can compute the IG attributions (see [notebook](#)) by doing:

```
from alibi.explainers import IntegratedGradients

ig = IntegratedGradients(model)  # TensorFlow model
result = ig.explain(
    scaler.transform(x),          # scaled data instance
    target=0,                    # model class probability prediction to obtain
    attribution for
)

plot_importance(result.data['attributions'][0], features, 0)
```

This gives:



Note 4: Comparison to ALE The alcohol feature value contributes negatively here to the “Good” prediction which seems to contradict the [ALE result](#). However, The instance x we choose has an alcohol content of 9.4%, which is reasonably low for a wine classed as “Good” and is consistent with the ALE plot. (The median for good wines is 10.8% and bad wines 9.7%)

Pros	Cons
Simple to understand and visualize, especially with image data	White-box method. Requires the partial derivatives of the model outputs with respect to inputs
Doesn't require access to the training data	Requires choosing the baseline which can have a significant effect on the outcome
<i>Satisfies several desirable properties</i>	

Kernel SHAP

Ex- plainer	Scop	Model types	Task types	Data types	Use	Re- sources
Kernel SHAP	Lo- cal	Black- box	Classification, Regression	Tabular (numeri- cal, categorical)	What does each feature contribute to the model prediction?	docs , paper

Kernel SHAP ([Alibi method docs](#)) is a method for computing the Shapley values of a model around an instance. [Shapley values](#) are a game-theoretic method of assigning payout to players depending on their contribution to an overall goal. In our case, the features are the players, and the payouts are the attributions.

Given any subset of features, we can ask how a feature's presence in that set contributes to the model output. We do this by computing the model output for the set with and without the specific feature. We obtain the Shapley value for that feature by considering these contributions with and without it present for all possible subsets of features.

Two problems arise. Most models are not trained to take a variable number of input features. And secondly, considering all possible sets of absent features leads to considering the [power set](#) which is prohibitively large when there are many features.

To solve the former, we sample from the **interventional conditional expectation**. This replaces missing features with values sampled from the training distribution. And to solve the latter, the kernel SHAP method samples on the space of subsets to obtain an estimate.

A downside of interfering in the distribution like this is that doing so introduces unrealistic samples if there are dependencies between the features.

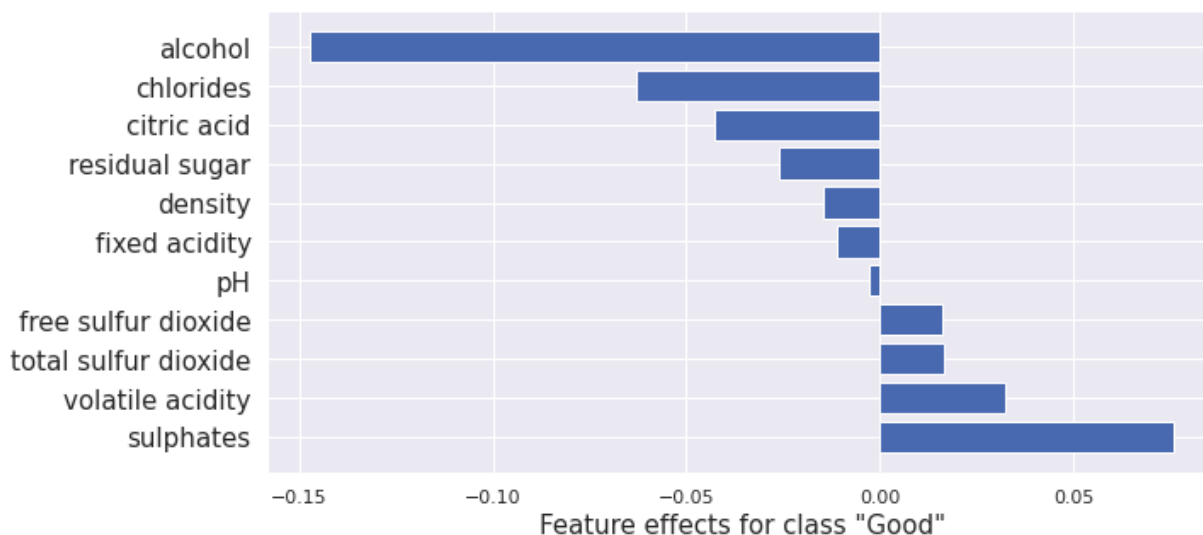
Alibi provides a wrapper to the [SHAP library](#). We can use this explainer to compute the Shapley values for a [sklearn random forest](#) model using the following (see [notebook](#)):

```
from alibi.explainers import KernelShap

# black-box model
predict_fn = lambda x: rfc.predict_proba(scaler.transform(x))
explainer = KernelShap(predict_fn, task='classification')
explainer.fit(X_train[0:100])
result = explainer.explain(x)

plot_importance(result.shap_values[1], features, 1)
```

This gives the following output:



This result is similar to the one for *Integrated Gradients* although there are differences due to using different methods and models in each case.

Pros	Cons
<i>Satisfies several desirable properties</i>	Kernel SHAP is slow owing to the number of samples required to estimate the Shapley values accurately
Shapley values can be easily interpreted and visualized	The interventional conditional probability introduces unrealistic data points
Very general as is a black-box method	Requires access to the training dataset

Path-dependent Tree SHAP

Explainer	Scope	Model types	Task types	Data types	Use	Resources
<i>Tree SHAP (path-dependent)</i>	Local	White-box (XGBoost, LightGBM, CatBoost, scikit-learn and pyspark tree models)	Classification, Regression	Tabular (numerical, categorical)	What does each feature contribute to the model prediction?	docs , paper

Computing the Shapley values for a model requires computing the interventional conditional expectation for each member of the [power set](#) of instance features. For tree-based models we can approximate this distribution by applying the tree as usual. However, for missing features, we take both routes down the tree, weighting each path taken by the proportion of samples from the training dataset that go each way. The tree SHAP method does this simultaneously for all members of the feature power set, obtaining a [significant speedup](#). Assume the random forest has T trees, with a depth of D , let L be the number of leaves and let M be the size of the feature set. If we compute the approximation for each member of the power set we obtain a time complexity of $O(TL2^M)$. In contrast, computing for all sets simultaneously we achieve $O(TLD^2)$.

To compute the path-dependent tree SHAP explainer for a random forest using Alibi (see [notebook](#)) we use:

```
from alibi.explainers import TreeShap
```

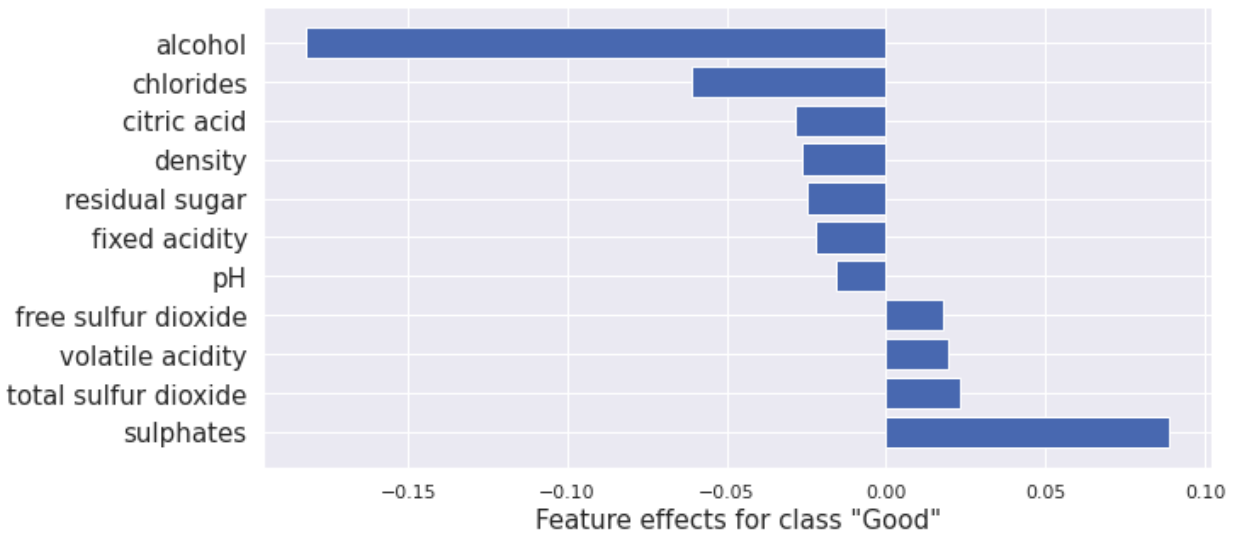
(continues on next page)

(continued from previous page)

```
# rfc is a random forest model
path_dependent_explainer = TreeShap(rfc)
path_dependent_explainer.fit()                                # path dependent TreeShap
↳ SHAP doesn't need any data

result = path_dependent_explainer.explain(scaler.transform(x)) # explain the scaled instance
↳ instance
plot_importance(result.shap_values[1], features, '"Good"')
```

From this we obtain:



This result is similar to the one for *Integrated Gradients* and *Kernel SHAP* although there are differences due to using different methods and models in each case.

Pros	Cons
<i>Satisfies several desirable properties</i>	Only applies to tree-based models
Very fast for a valuable category of models	Uses an approximation of the interventional conditional expectation instead of computing it directly
Doesn't require access to the training data	
Shapley values can be easily interpreted and visualized	

Interventional Tree SHAP

Explainer	Scope	Model types	Task types	Data types	Use	Resources
<i>Tree SHAP (interventional)</i>	Local	White-box (<i>XGBoost</i> , <i>LightGBM</i> , <i>CatBoost</i> , <i>scikit-learn</i> and <i>pyspark tree models</i>)	Classification, Regression	Tabular (numerical, categorical)	What does each feature contribute to the model prediction?	docs , paper

Suppose we sample a reference data point, r , from the training dataset. Let F be the set of all features. For each feature, i , we then enumerate over all subsets of $S \subset F \setminus \{i\}$. If a subset is missing a feature, we replace it with the corresponding one in the reference sample. We can then compute $f(S)$ directly for each member of the power set of instance features to get the Shapley values.

Enforcing independence of the S and $F \setminus S$ in this way is known as intervening in the underlying data distribution and is the source of the algorithm's name. Note that this breaks any independence between features in the dataset, which means the data points we're sampling won't always be realistic.

For a single tree and sample r if we iterate over all the subsets of $S \subset F \setminus \{i\}$, it would give $O(M2^M)$ time complexity. The interventional tree SHAP algorithm runs with $O(TLD)$ time complexity.

The main difference between the interventional and the path-dependent tree SHAP methods is that the latter approximates the interventional conditional expectation, whereas the former method calculates it directly.

To compute the interventional tree SHAP explainer for a random forest using Alibi (see [notebook](#)), we use:

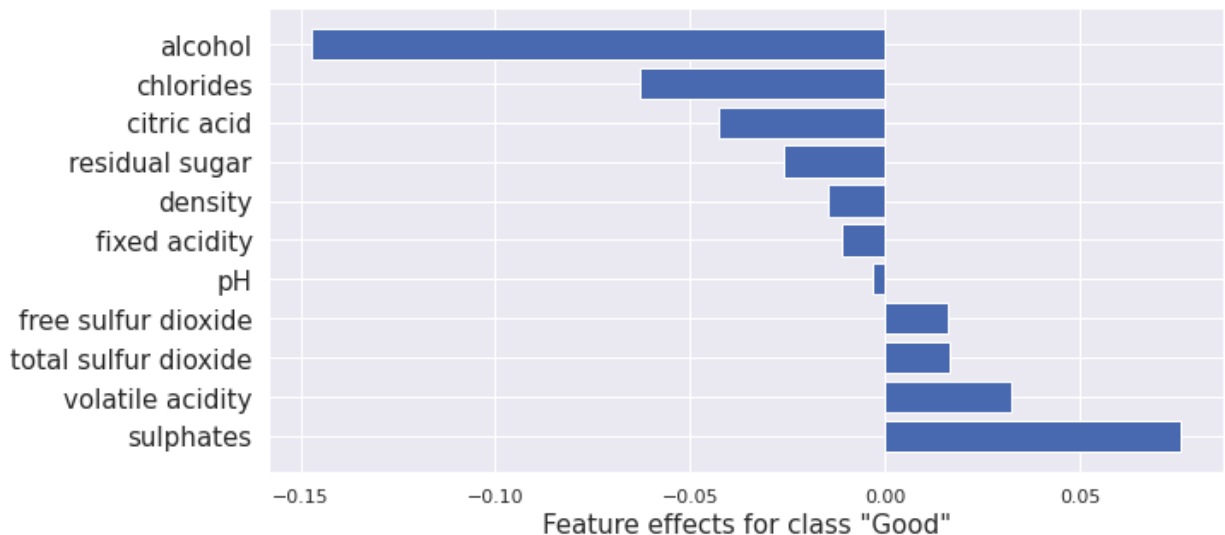
```
from alibi.explainers import TreeShap

# rfc is a random forest classifier model
tree_explainer_interventional = TreeShap(rfc)

# interventional tree SHAP is slow for large datasets so we take first 100 samples of
# training data.
tree_explainer_interventional.fit(scaler.transform(X_train[0:100]))

result = tree_explainer_interventional.explain(scaler.transform(x)) # explain the
# scaled instance
plot_importance(result.shap_values[1], features, "Good")
```

From this we obtain:



This result is similar to the one for *Integrated Gradients*, *Kernel SHAP*, *Path-dependent Tree SHAP* although there are differences due to using different methods and models in each case.

For a great interactive explanation of the interventional Tree SHAP method see.

Pros	Cons
<i>Satisfies several desirable properties</i>	Only applies to tree-based models
Very fast for a valuable category of models	Requires access to the dataset
Shapley values can be easily interpreted and visualized	Typically slower than the path-dependent method
Computes the interventional conditional expectation exactly unlike the path-dependent method	

1.2.4 4. Counterfactual instances

Given an instance of the dataset and a prediction given by a model, a question naturally arises how would the instance minimally have to change for a different prediction to be provided. Such a generated instance is known as a *counterfactual*. Counterfactuals are local explanations as they relate to a single instance and model prediction.

Given a classification model trained on the MNIST dataset and a sample from the dataset, a counterfactual would be a generated image that closely resembles the original but is changed enough that the model classifies it as a different number from the original instance.

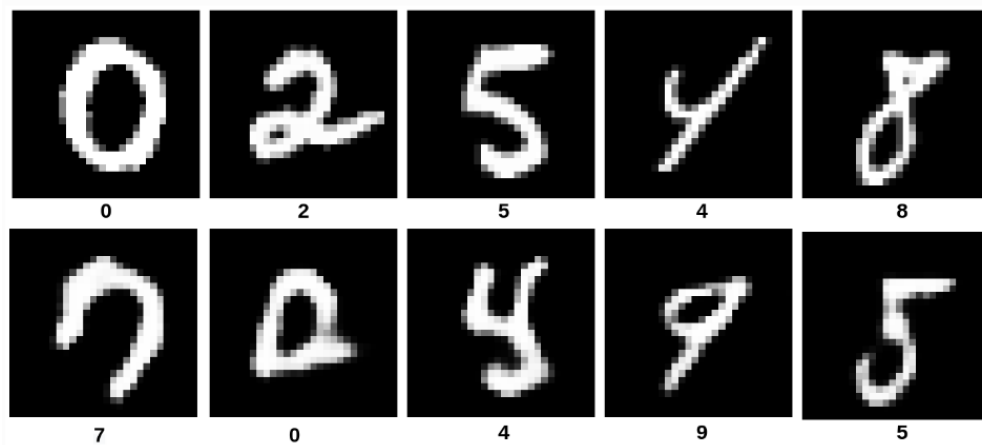


Fig. 2: From Samoilescu RF et al., *Model-agnostic and Scalable Counterfactual Explanations via Reinforcement Learning*, 2021

Counterfactuals can be used to both [debug](#) and [augment](#) model functionality. Given tabular data that a model uses to make financial decisions about a customer, a counterfactual would explain how to change their behavior to obtain a different conclusion. Alternatively, it may tell the Machine Learning Engineer that the model is drawing incorrect assumptions if the recommended changes involve features that are irrelevant to the given decision. However, practitioners must still be wary of [bias](#).

A counterfactual, x_{cf} , needs to satisfy

- The model prediction on x_{cf} needs to be close to the pre-defined output (e.g. desired class label).
- The counterfactual x_{cf} should be interpretable.

The first requirement is clear. The second, however, requires some idea of what interpretable means. Alibi exposes four methods for finding counterfactuals: *counterfactual instances (CFI)*, *contrastive explanations (CEM)*, *counterfactuals guided by prototypes (CFP)*, and *counterfactuals with reinforcement learning (CFRL)*. Each of these methods deals with interpretability slightly differently. However, all of them require sparsity of the solution. This means we prefer to only change a small subset of the features which limits the complexity of the solution making it more understandable.

Note that sparse changes to the instance of interest doesn't guarantee that the generated counterfactual is believably a member of the data distribution. **CEM**, **CFP**, and **CFRL** also require that the counterfactual be in distribution in order to be interpretable.

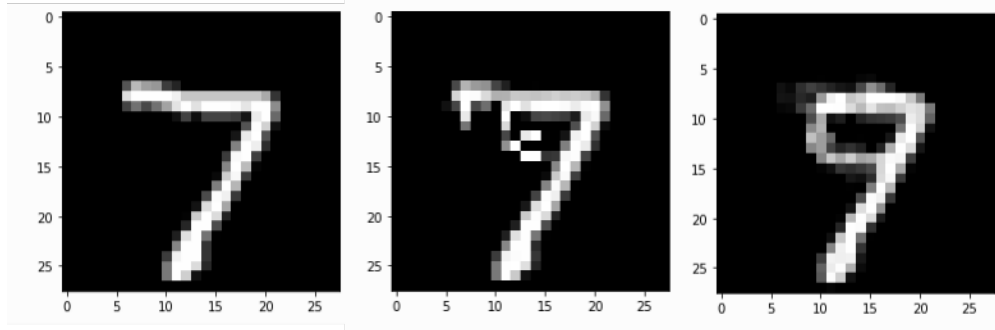


Fig. 3: *Original MNIST 7 instance, Counterfactual instances constructed using 1) **counterfactual instances method**, 2) **counterfactual instances with prototypes method***

The first three methods **CFI**, **CEM**, **CFP** all construct counterfactuals using a very similar method. They build them by defining a loss that prefer interpretable instances close to the target class. They then use gradient descent to move within the feature space until they obtain a counterfactual of sufficient quality. The main difference is the **CEM** and **CFP** methods also train an autoencoder to ensure that the constructed counterfactuals are within the data-distribution.

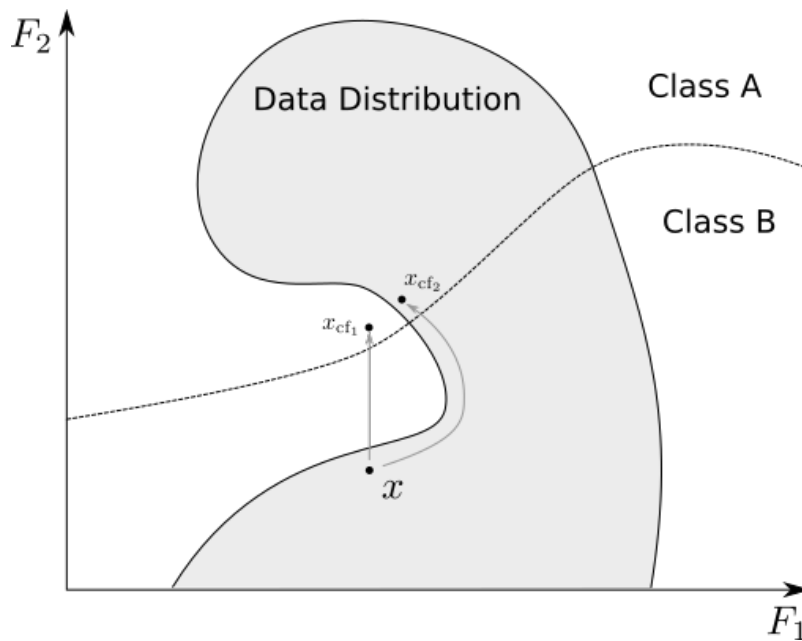


Fig. 4: *Obtaining counterfactuals using gradient descent with and without autoencoder trained on data distribution*

These three methods only realistically work for grayscale images and anything multi-channel will not be interpretable. In order to get quality results for multi-channel images practitioners should use **CFRL**.

CFRL uses a similar loss to CEM and CFP but applies reinforcement learning to train a model which will generate counterfactuals on demand.

Note 5: fit and explain method runtime differences

Alibi explainers expose two methods, `fit` and `explain`. Typically in machine learning the method that takes the most time is the `fit` method, as that's where the model optimization conventionally takes place. In explainability, the `explain` step often requires the bulk of computation. However, this isn't always the case.

Among the explainers in this section, there are two approaches taken. The first finds a counterfactual when the user requests the insight. This happens during the `.explain()` method call on the explainer class. This is done by running gradient descent on model inputs to find a counterfactual. The methods that take this approach are **counterfactual instances**, **contrastive explanation**, and **counterfactuals guided by prototypes**. Thus, the `fit` method in these cases is quick, but the `explain` method is slow.

The other approach, **counterfactuals with reinforcement learning**, trains a model that produces explanations on demand. The training takes place during the `fit` method call, so this has a long runtime while the `explain` method is quick. If you want performant explanations in production environments, then the latter approach is preferable.

Counterfactual Instances

Explainer	Scope	Model types	Task types	Data types	Use	Re-sources
<i>Counterfactual Instances</i>	Local	Black-box (<i>differentiable</i>), White-box (<i>TensorFlow</i>)	Classification	Tabular(numerical) Image	What minimal change to features is required to reclassify the current prediction?	<i>docs</i> , <i>paper</i>

Let the model be given by f , and let p_t be the target probability of class t . Let λ be a hyperparameter. This method constructs counterfactual instances from an instance X by running gradient descent on a new instance X' to minimize the following loss:

$$L(X', X) = (f_t(X') - p_t)^2 + \lambda L_1(X', X)$$

The first term pushes the constructed counterfactual towards the desired class, and the use of the L_1 norm encourages sparse solutions.

This method requires computing gradients of the loss in the model inputs. If we have access to the model and the gradients are available, this can be done directly. If not, we can use numerical gradients, although this comes at a considerable performance cost.

A problem arises here in that encouraging sparse solutions doesn't necessarily generate interpretable counterfactuals. This happens because the loss doesn't prevent the counterfactual solution from moving off the data distribution. Thus, you will likely get an answer that doesn't look like something that you would expect to see from the data.

To use the counterfactual instances method from Alibi applied to the wine quality dataset (see [notebook](#)), use:

```
from alibi.explainers import Counterfactual

explainer = Counterfactual(
    model,                                # The model to explain
    shape=(1,) + X_train.shape[1:],      # The shape of the model input
    target_proba=0.51,                    # The target class probability
    tol=0.01,                             # The tolerance for the loss
    target_class='other',                 # The target class to obtain
)

result_cf = explainer.explain(scaler.transform(x))
```

(continues on next page)

(continued from previous page)

```
print("Instance prediction:", model.predict(scaler.transform(x))[0].argmax())
print("Counterfactual prediction:", model.predict(result_cf.data['cf']['X'])[0].argmax())
```

Gives the expected result:

```
Instance prediction: 0      # "good"
Counterfactual prediction: 1 # "bad"
```

Pros	Cons
Both a black-box and white-box method	Not likely to give human interpretable instances
Doesn't require access to the training dataset	Requires tuning of λ hyperparameter
	Slow for black-box models due to having to numerically evaluate gradients

Contrastive Explanation Method (Pertinent Negatives)

Explainer	Scope	Model types	Task types	Data types	Use	Resources
<i>Contrastive Explanation Method</i>	Local	Black-box (<i>differentiable</i>), White-box (<i>TensorFlow</i>)	Classification	Tabular (numerical), Image	What minimal change to features is required to reclassify the current prediction?	docs , paper

CEM follows a similar approach to the above but includes three new details. Firstly an elastic net $\beta L_1 + L_2$ regularizer term is added to the loss. This term causes the solutions to be both close to the original instance and sparse.

Secondly, we require that δ only adds new features rather than takes them away. We need to define what it means for a feature to be present so that the perturbation only works to add and not remove them. In the case of the MNIST dataset, an obvious choice of “present” feature is if the pixel is equal to 1 and absent if it is equal to 0. This is simple in the case of the MNIST data set but more difficult in complex domains such as colour images.

Thirdly, by training an autoencoder to penalize counterfactual instances that deviate from the data distribution. This works by minimizing the reconstruction loss of the autoencoder applied to instances. If a generated instance is unlike anything in the dataset, the autoencoder will struggle to recreate it well, and its loss term will be high. We require three hyperparameters c , β and γ to define the following loss:

$$L = c \cdot L_{pred}(\delta) + \beta L_1(\delta, x) + L_2^2(\delta, x) + \gamma \|\delta - AE(\delta)\|_2^2$$

A subtle aspect of this method is that it requires defining the absence or presence of features as delta is restrained only to allow you to add information. For the MNIST digits, it's reasonable to assume that the black background behind each written number represents an absence of information. Similarly, in the case of colour images, you might take the median pixel value to convey no information, and moving away from this value adds information. For numerical tabular data, we can use the feature mean. In general, choosing a non-informative value for each feature is non-trivial, and domain knowledge is required. This is the reverse process to the *contrastive explanation method (pertinent-positives)* method introduced in the section on local necessary features in which we take away features rather than add them.

This approach extends the definition of interpretable to include a requirement that the computed counterfactual be believably a member of the dataset. This isn't always satisfied (see image below). In particular, the constructed counterfactual often doesn't look like a member of the target class.

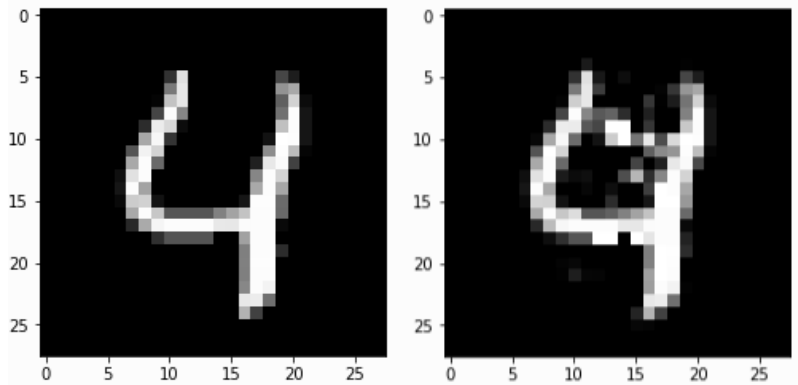


Fig. 5: An original MNIST instance and a pertinent negative obtained using CEM.

To compute a pertinent-negative using Alibi (see [notebook](#)) we use:

```
from alibi.explainers import CEM

cem = CEM(model,                                # model to explain
          shape=(1,) + X_train.shape[1:],       # shape of the model input
          mode='PN',                             # pertinent negative mode
          kappa=0.2,                             # Confidence parameter for the attack loss
          ↪ term
          beta=0.1,                             # Regularization constant for L1 loss term
          ae_model=ae                           # autoencoder model
        )

cem.fit(
    scaler.transform(X_train), # scaled training data
    no_info_type='median'     # non-informative value for each feature
)
result_cem = cem.explain(scaler.transform(x), verbose=False)
cem_cf = result_cem.data['PN']

print("Instance prediction:", model.predict(scaler.transform(x))[0].argmax())
print("Counterfactual prediction:", model.predict(cem_cf)[0].argmax())
```

Gives the expected result:

```
Instance prediction: 0      # "good"
Counterfactual prediction: 1 # "bad"
```

This method can apply to both black-box and white-box models. There is a performance cost from computing the numerical gradients in the black-box case due to having to numerically evaluate gradients.

Pros	Cons
Provides more interpretable instances than the counterfactual instances' method.	Requires access to the dataset to train the autoencoder
Applies to both white and black-box models	Requires setup and configuration in choosing c , γ and β
	Requires training an autoencoder
	Requires domain knowledge when choosing what it means for a feature to be present or not
	Slow for black-box models

Counterfactuals Guided by Prototypes

Explainer	Scope	Model types	Task types	Data types	Use	Resources
<i>Counterfactuals Guided by Prototypes</i>	Local	Black-box (<i>differentiable</i>), White-box (<i>TensorFlow</i>)	Classification	Tabular (numerical, categorical), Image	What minimal change to features is required to reclassify the current prediction?	docs , paper

For this method, we add another term to the loss that optimizes for the distance between the counterfactual instance and representative members of the target class. In doing this, we require interpretability also to mean that the generated counterfactual is believably a member of the target class and not just in the data distribution.

With hyperparameters c , γ and β , the loss is given by:

$$L(X'|X) = c \cdot L_{pred}(X') + \beta L_1(X', X) + L_2(X', X)^2 + \gamma L_2(X', AE(X'))^2 + L_2(X', X_{proto})$$

This method produces much more interpretable results than *CFI* and *CEM*.

Because the prototype term steers the solution, we can remove the prediction loss term. This makes this method much faster if we are using a black-box model as we don't need to compute the gradients numerically. However, occasionally the prototype isn't a member of the target class. In this case you'll end up with an incorrect counterfactual.

To use the counterfactual with prototypes method in Alibi (see [notebook](#)) we do:

```
from alibi.explainers import CounterfactualProto

explainer = CounterfactualProto(
    model,                                # The model to explain
    shape=(1,) + X_train.shape[1:],      # shape of the model input
    ae_model=ae,                          # The autoencoder
    enc_model=ae.encoder                  # The encoder
)

explainer.fit(scaler.transform(X_train)) # Fit the explainer with scaled data

result_proto = explainer.explain(scaler.transform(x), verbose=False)

proto_cf = result_proto.data['cf']['X']
```

(continues on next page)

(continued from previous page)

```
print("Instance prediction:", model.predict(scaler.transform(x))[0].argmax())
print("Counterfactual prediction:", model.predict(proto_cf)[0].argmax())
```

We get the following results:

```
Instance prediction: 0      # "good"
Counterfactual prediction: 1 # "bad"
```

Pros	Cons
Generates more interpretable instances than the CEM method	Requires access to the dataset
Black-box version of the method is fast	Requires setup and configuration in choosing γ , β and c
Applies to more data-types	Requires training an autoencoder

Counterfactuals with Reinforcement Learning

Explainer	Scope	Model types	Task types	Data types	Use	Resources
<i>Counterfactuals with Reinforcement Learning</i>	Local	Black-box	Classification	Tabular (numerical, categorical), Image	What minimal change to features is required to reclassify the current prediction?	<i>docs, paper</i>

This black-box method splits from the approach taken by the above three significantly. Instead of minimizing a loss during the explain method call, it trains a **new model** when **fitting** the explainer called an **actor** that takes instances and produces counterfactuals. It does this using **reinforcement learning**. In reinforcement learning, an actor model takes some state as input and generates actions; in our case, the actor takes an instance with a target classification and attempts to produce a member of the target class. Outcomes of actions are assigned rewards dependent on a reward function designed to encourage specific behaviors. In our case, we reward correctly classified counterfactuals generated by the actor. As well as this, we reward counterfactuals that are close to the data distribution as modeled by an autoencoder. Finally, we require that they are sparse perturbations of the original instance. The reinforcement training step pushes the actor to take high reward actions. CFRL is a black-box method as the process by which we update the actor to maximize the reward only requires estimating the reward via sampling the counterfactuals.

As well as this, CFRL actors can be trained to ensure that certain **constraints** can be taken into account when generating counterfactuals. This is highly desirable as a use case for counterfactuals is to suggest the necessary changes to an instance to obtain a different classification. In some cases, you want these changes to be constrained, for instance, when dealing with immutable characteristics. In other words, if you are using the counterfactual to advise changes in behavior, you want to ensure the changes are enactable. Suggesting that someone needs to be two years younger to apply for a loan isn't very helpful.

The training process requires randomly sampling data instances, along with constraints and target classifications. We can then compute the reward and update the actor to maximize it. We do this without needing access to the model internals; we only need to obtain a prediction in each case. The end product is a model that can generate interpretable counterfactual instances at runtime with arbitrary constraints.

To use CFRL on the wine dataset (see [notebook](#)), we use:

```
from alibi.explainers import CounterfactualRL

predict_fn = lambda x: model(x)

cfrl_explainer = CounterfactualRL(
    predictor=predict_fn,          # The model to explain
    encoder=ae.encoder,           # The encoder
    decoder=ae.decoder,           # The decoder
    latent_dim=7,                 # The dimension of the autoencoder latent space
    coeff_sparsity=0.5,           # The coefficient of sparsity
    coeff_consistency=0.5,        # The coefficient of consistency
    train_steps=10000,            # The number of training steps
    batch_size=100,               # The batch size
)

cfrl_explainer.fit(X=scaler.transform(X_train))

result_cfrl = cfrl_explainer.explain(X=scaler.transform(x), Y_t=np.array([1]))
print("Instance prediction:", model.predict(scaler.transform(x))[0].argmax())
print("Counterfactual prediction:", model.predict(result_cfrl.data['cf']['X'])[0].
      ↪argmax())
```

Which gives the following output:

```
Instance prediction: 0          # "good"
Counterfactual prediction: 1    # "bad"
```

Note 6: CFRL explainer

Alibi exposes two explainer methods for counterfactuals with reinforcement learning. The first is the CounterfactualRL and the second is CounterfactualRITabular. The difference is that CounterfactualRITabular is designed to support categorical features. See the [CFRL documentation page](#) for more details.

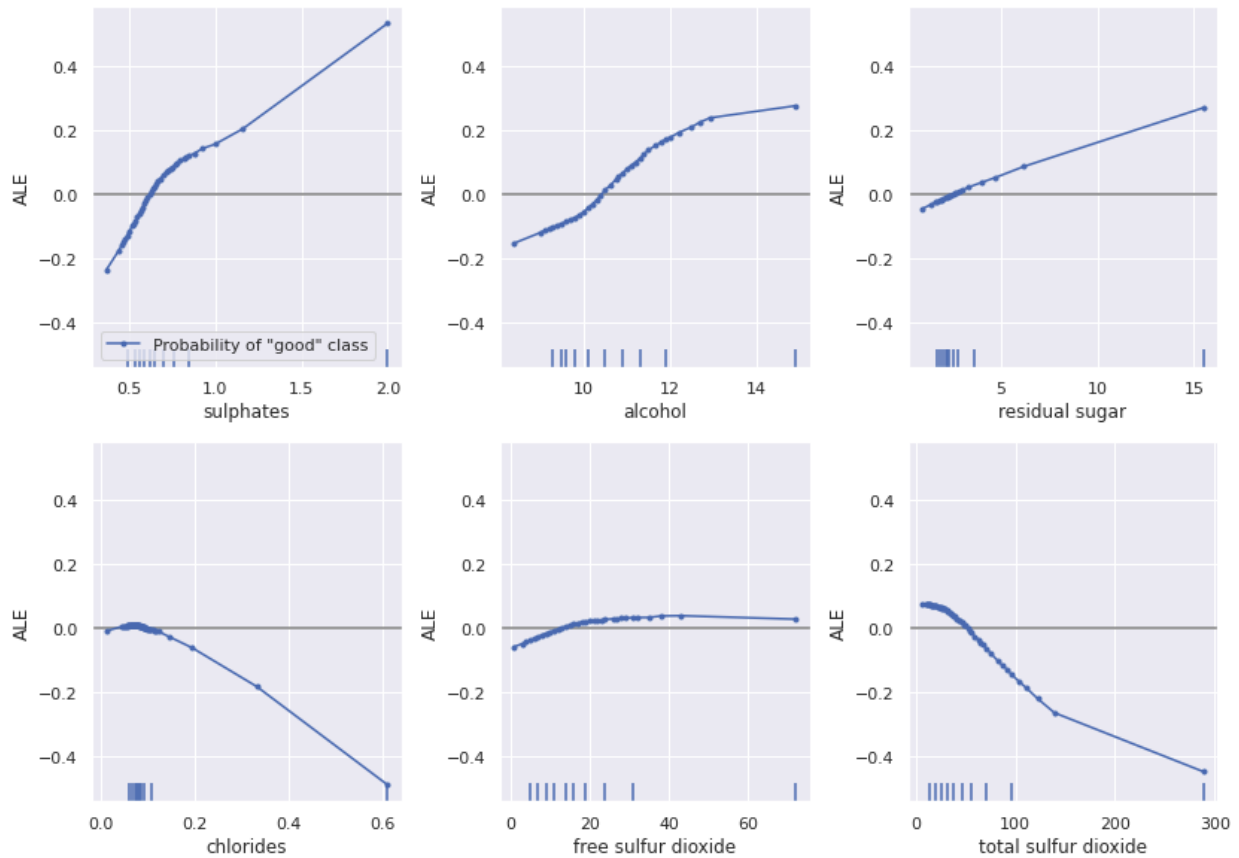
Pros	Cons
Generates more interpretable instances than the CEM method	Longer to fit the model
Very fast at runtime	Requires to fit an autoencoder
Can be trained to account for arbitrary constraints	Requires access to the training dataset
General as is a black-box algorithm	

Counterfactual Example Results

For each of the four explainers, we have generated a counterfactual instance. We compare the original instance to each:

Feature	Instance	CFI	CEM	CFP	CFRL
sulphates	0.67	0.64	0.549	0.623	0.598
alcohol	10.5	9.88	9.652	9.942	9.829
residual sugar	1.6	1.582	1.479	1.6	2.194
chlorides	0.062	0.061	0.057	0.062	0.059
free sulfur dioxide	5.0	4.955	2.707	5.0	6.331
total sulfur dioxide	12.0	11.324	12.0	12.0	14.989
fixed acidity	9.2	9.23	9.2	9.2	8.965
volatile acidity	0.36	0.36	0.36	0.36	0.349
citric acid	0.34	0.334	0.34	0.34	0.242
density	0.997	0.997	0.997	0.997	0.997
pH	3.2	3.199	3.2	3.2	3.188

The CFI, CEM, and CFRL methods all perturb more features than CFP, making them less interpretable. Looking at the ALE plots, we can see how the counterfactual methods change the features to flip the prediction. In general, each method seems to decrease the sulphates and alcohol content to obtain a “bad” classification consistent with the ALE plots. Note that the ALE plots potentially miss details local to individual instances as they are global insights.



1.2.5 5. Similarity explanations

Ex- plainer	Scop	Model types	Task types	Data types	Use	Re- sources
<i>Simi- larity explana- tions</i>	Lo- cal	White- box	Classi- fication, Regres- sion	Tabular (numerical, categorical), Text and Image	What are the instances in the training set that are most similar to the instance of interest according to the model?	<i>docs</i> , <i>pa- per</i>

Similarity explanations are instance-based explanations that focus on training data points to justify a model prediction on a test instance. Given a trained model and a test instance whose prediction is to be explained, these methods scan the training set, finding the most similar data points according to the model which forms an explanation. This type of explanation can be interpreted as the model justifying its prediction by referring to similar instances which may share the same prediction—*“I classify this image as a ‘Golden Retriever’ because it is most similar to images in the training set which I also classified as ‘Golden Retriever’”*.

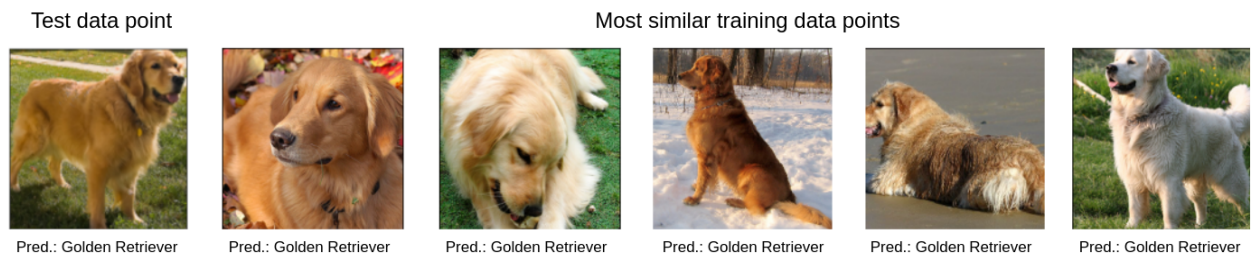


Fig. 6: A similarity explanation justifies the classification of an image as a ‘Golden Retriever’ because most similar instances in the training set are also classified as ‘Golden Retriever’.

GETTING STARTED

2.1 Installation

Alibi works with Python 3.7+ and can be installed from [PyPI](#) or [conda-forge](#) by following the instructions below.

Install via PyPI

- Alibi can be installed from [PyPI](#) with `pip`:

Standard

Default installation.

```
pip install alibi
```

SHAP

Installation with support for computing [SHAP](#) values.

```
pip install alibi[shap]
```

Distributed

Installation with support for *distributed Kernel SHAP*.

```
pip install alibi[ray]
```

TensorFlow

Installation with support for tensorflow backends. Required for

- *Contrastive Explanation Method (CEM)*
- *Counterfactuals Guided by Prototypes*
- *Counterfactual Instances*
- *Integrated gradients*

- *Anchors on Textual data* with `sampling_strategy='language_model'`
- One of Torch or TensorFlow is required for the *Counterfactuals with RL* methods

```
pip install alibi[tensorflow]
```

Torch

Installation with support for torch backends. One of Torch or TensorFlow is required for:

- *Counterfactuals with RL*
- *Similarity explanations*

```
pip install alibi[torch]
```

All

Installs all optional dependencies.

```
pip install alibi[all]
```

Install via conda-forge

- To install the conda-forge version it is recommended to use [mamba](#), which can be installed to the *base* conda environment with:

```
conda install mamba -n base -c conda-forge
```

- [mamba](#) can then be used to install alibi in a conda environment:

Standard

Default installation.

```
mamba install -c conda-forge alibi
```

SHAP

Installation with support for computing [SHAP](#) values.

```
mamba install -c conda-forge alibi shap
```

Distributed

Installation with support for distributed computation of explanations.

```
mamba install -c conda-forge alibi ray
```

2.2 Features

Alibi is a Python package designed to help explain the predictions of machine learning models and gauge the confidence of predictions. The focus of the library is to support the widest range of models using black-box methods where possible.

To get a list of the latest available model explanation algorithms, you can type:

```
import alibi
alibi.explainers.__all__
```

```
['ALE',
 'AnchorTabular',
 'DistributedAnchorTabular',
 'AnchorText',
 'AnchorImage',
 'CEM',
 'Counterfactual',
 'CounterfactualProto',
 'CounterfactualRL',
 'CounterfactualRLTabular',
 'PartialDependence',
 'TreePartialDependence',
 'PartialDependenceVariance',
 'PermutationImportance',
 'plot_ale',
 'plot_pd',
 'plot_pd_variance',
 'plot_permutation_importance',
 'IntegratedGradients',
 'KernelShap',
 'TreeShap',
 'GradientSimilarity']
```

For gauging model confidence:

```
alibi.confidence.__all__
```

```
['linearity_measure',
 'LinearityMeasure',
 'TrustScore']
```

For dataset summarization

```
alibi.prototypes.__all__
```

```
['ProtoSelect',  
 'visualize_image_prototypes']
```

For detailed information on the methods:

- *Overview of available methods*
 - *Accumulated Local Effects*
 - *Anchor explanations*
 - *Contrastive Explanation Method (CEM)*
 - *Counterfactual Instances*
 - *Counterfactuals Guided by Prototypes*
 - *Counterfactuals with RL*
 - *Integrated gradients*
 - *Kernel SHAP*
 - *Linearity Measure*
 - *ProtoSelect*
 - *PartialDependence*
 - *PD Variance*
 - *Permutation Importance*
 - *TreeShap*
 - *Trust Scores*
 - *Similarity explanations*

2.3 Basic Usage

The alibi explanation API takes inspiration from `scikit-learn`, consisting of distinct initialize, fit and explain steps. We will use the *Anchor method on tabular data* to illustrate the API.

First, we import the explainer:

```
from alibi.explainers import AnchorTabular
```

Next, we initialize it by passing it a *prediction function* and any other necessary arguments:

```
explainer = AnchorTabular(predict_fn, feature_names)
```

Some methods require an additional `.fit` step which requires access to the training set the model was trained on:

```
explainer.fit(X_train)
```

```
AnchorTabular(meta={  
    'name': 'AnchorTabular',  
    'type': ['blackbox'],  
    'explanations': ['local'],
```

(continues on next page)

(continued from previous page)

```
'params': {'seed': None, 'disc_perc': (25, 50, 75)}
})
```

Finally, we can call the explainer on a test instance which will return an `Explanation` object containing the explanation and any additional metadata returned by the computation:

```
explanation = explainer.explain(x)
```

The returned `Explanation` object has `meta` and `data` attributes which are dictionaries containing any explanation metadata (e.g. parameters, type of explanation) and the explanation itself respectively:

```
explanation.meta
```

```
{'name': 'AnchorTabular',
 'type': ['blackbox'],
 'explanations': ['local'],
 'params': {'seed': None,
 'disc_perc': (25, 50, 75),
 'threshold': 0.95,
 'delta': ...truncated output...
```

```
explanation.data
```

```
{'anchor': ['petal width (cm) > 1.80', 'sepal width (cm) <= 2.80'],
 'precision': 0.9839228295819936,
 'coverage': 0.31724137931034485,
 'raw': {'feature': [3, 1],
 'mean': [0.6453362255965293, 0.9839228295819936],
 'precision': [0.6453362255965293, 0.9839228295819936],
 'coverage': [0.20689655172413793, 0.31724137931034485],
 'examples': ...truncated output...
```

The top level keys of both `meta` and `data` dictionaries are also exposed as attributes for ease of use of the explanation:

```
explanation.anchor
```

```
['petal width (cm) > 1.80', 'sepal width (cm) <= 2.80']
```

Some algorithms, such as *Kernel SHAP*, can run batches of explanations in parallel, if the number of cores is specified in the algorithm constructor:

```
distributed_ks = KernelShap(predict_fn, distributed_opts={'n_cpus': 10})
```

Note that this requires the user to run `pip install alibi[ray]` to install dependencies of the distributed backend.

The exact details will vary slightly from method to method, so we encourage the reader to become familiar with the *types of algorithms supported* in Alibi.

ALGORITHM OVERVIEW

This page provides a high-level overview of the algorithms and their features currently implemented in Alibi.

3.1 Model Explanations

These algorithms provide **instance-specific** (sometimes also called **local**) explanations of ML model predictions. Given a single instance and a model prediction they aim to answer the question “Why did my model make this prediction?” Most of the following algorithms work with **black-box** models meaning that the only requirement is to have access to a prediction function (which could be an API endpoint for a model in production). For an extended discussion see [*White-box and black-box models*](#).

The following table summarizes the capabilities of the current algorithms:

Method	Models	Exp. types	Classification	Regression	Tabular	Text	Image	Cat. data	Train	Dist.
<i>ALE</i>	BB	global	✓	✓	✓					
<i>Partial Dependence</i>	BB WB	global	✓	✓	✓			✓		
<i>PD Variance</i>	BB WB	global	✓	✓	✓			✓		
<i>Permutation Importance</i>	BB	global	✓	✓	✓			✓		
<i>Anchors</i>	BB	local	✓		✓	✓	✓	✓	For Tabular	
<i>CEM</i>	BB* TF/Keras	local	✓		✓		✓		Optional	
<i>Counterfactuals</i>	BB* TF/Keras	local	✓		✓		✓		No	
<i>Prototype Counterfactuals</i>	BB* TF/Keras	local	✓		✓		✓	✓	Optional	
<i>Counterfactuals with RL</i>	BB	local	✓		✓		✓	✓	✓	
<i>Integrated Gradients</i>	TF/Keras	local	✓	✓	✓	✓	✓	✓	Optional	
<i>Kernel SHAP</i>	BB	local global	✓	✓	✓			✓	✓	✓
<i>Tree SHAP</i>	WB	local global	✓	✓	✓			✓	Optional	
<i>Similarity explanations</i>	WB	local	✓	✓	✓	✓	✓	✓	✓	

Key:

- **BB** - *black-box* (only require a prediction function)
- **BB*** - *black-box* but assume model is differentiable
- **WB** - requires *white-box* model access. There may be limitations on models supported
- **TF/Keras** - TensorFlow models via the Keras API
- **Local** - instance specific explanation, why was this prediction made?
- **Global** - explains the model with respect to a set of instances
- **Cat. data** - support for categorical features
- **Train** - whether a training set is required to fit the explainer
- **Dist.** - whether a batch of explanations can be executed in parallel

Accumulated Local Effects (ALE): calculates first-order feature effects on the model with respect to a dataset. Intended for use on tabular datasets, currently supports numerical features. [Documentation](#), [regression example](#), [classification example](#).

Partial Dependence: computes the marginal effect that one or multiple features have on the predicted outcome of a model with respect to a dataset. Intended for use on tabular datasets, black-box and white-box (scikit-learn) models, supporting numerical and categorical features. [Documentation](#), [Bike rental](#).

Partial Dependence Variance: computes the global feature importance or the feature interaction of a pair of features. The feature importance and the feature interactions are summarized in a single positive number given by the variance within the Partial Dependence function. Intended for use on tabular datasets, black-box and white-box (scikit-learn) models, supporting numerical and categorical features. [Documentation](#), [Friedman’s regression problem](#).

Permutation Importance: computes the global feature importance. The computation of the feature importance is based on the degree of model performance degradation when the feature values within a feature column are permuted. Intended for use on tabular dataset, black-box models, supporting numerical and categorical features. [Documentation](#), [Who’s Going to Leave Next?](#).

Anchor Explanations: produce an “anchor” - a small subset of features and their ranges that will almost always result in the same model prediction. [Documentation](#), [tabular example](#), [text classification](#), [image classification](#).

Contrastive Explanation Method (CEM): produce a pertinent positive (PP) and a pertinent negative (PN) instance. The PP instance finds the features that should be minimally and sufficiently present to predict the same class as the original prediction (a PP acts as the “most compact” representation of the instance to keep the same prediction). The PN instance identifies the features that should be minimally and necessarily absent to maintain the original prediction (a PN acts as the closest instance that would result in a different prediction). [Documentation](#), [tabular example](#), [image classification](#).

Counterfactual Explanations: generate counterfactual examples using a simple loss function. [Documentation](#), [image classification](#).

Counterfactual Explanations Guided by Prototypes: generate counterfactuals guided by nearest class prototypes other than the class predicted on the original instance. It can use both an encoder or k-d trees to define the prototypes. This method can speed up the search, especially for black box models, and create interpretable counterfactuals. [Documentation](#), [tabular example](#), [tabular example with categorical features](#), [image classification](#).

Model-agnostic Counterfactual Explanations with RL: transform the optimization procedure into an end-to-end learnable process, allowing to generate batches of counterfactual instances in a single forward pass. The method is model-agnostic (does not assume differentiability) and relies only on feedback from model predictions, allows for generating target-conditional counterfactual instances, flexible feature range constraints for numerical and categorical attributes, including the immutability of protected features (e.g. gender, race) and can be easily extended to other data modalities such as images. [Documentation](#), [tabular_example](#), [image_classification](#).

Integrated gradients: attribute an importance score to each element of the input or an internal layer of the the model with respect to a given baseline. The attributions are calculated as the path integral of the model gradients along a straight line from the baseline to the input. [Documentation](#), [MNIST example](#), [Imagenet example](#), [IMDB example](#), [Transformers example](#).

Kernel Shapley Additive Explanations (Kernel SHAP): attribute the change of a model output with respect to a given baseline (e.g., average over a reference set) to each of the input features. This is achieved for each feature in turn, by averaging the difference in the model output observed when the feature whose contribution is to be estimated is part of a group of “present” input features and the value observed when the feature is excluded from said group. The features that are not “present” (i.e., are missing) are replaced with values from a background dataset. This algorithm can be used to explain regression models and it is optimised to distribute batches of explanations. [Documentation](#), [continuous data](#), [more continuous data](#), [categorical data](#), [distributed_batch_explanations](#).

Tree Shapley Additive Explanations (Tree SHAP): attribute the change of a model output with respect to a baseline (e.g., average over a reference set or inferred from node data) to each of the input features. Similar to Kernel SHAP, the shap value of each feature is computed by averaging the difference of the model output observed when the feature is part of a group of “present” features and when the feature is excluded from said group, over all possible subsets of “present” features. Different estimation procedures for the effect of selecting different subsets of “present” features on the model output give rise to the interventional feature perturbation and the path-dependent feature perturbation variants of Tree SHAP. This algorithm can be used to explain regression models. [Documentation](#), [interventional feature perturbation Tree SHAP](#), [path-dependent feature perturbation Tree SHAP](#).

Similarity explanations: present instances in the training set that are similar to the instance of interest according to a kernel metric. The implemented kernels are gradient-based, meaning that the similarity between 2 instances is

based on the gradients of the loss function with respect to the model’s parameters calculated at each of the instances. [Documentation](#), [MNIST example](#), [Imagenet example](#), [20 news groups example](#).

3.2 Model Confidence

These algorithms provide **instance-specific** scores measuring the model confidence for making a particular prediction.

Method	Mod-els	Classifi-cation	Regres-sion	Tabu-lar	Text	Im-ages	Categorical Features	Train set re-quired
<i>Trust Scores</i>	BB	✓		✓	✓ ¹	✓ ²		Yes
<i>Linearity Measure</i>	BB	✓	✓	✓		✓		Optional

Trust scores: produce a “trust score” of a classifier’s prediction. The trust score is the ratio between the distance to the nearest class different from the predicted class and the distance to the predicted class, higher scores correspond to more trustworthy predictions. [Documentation](#), [tabular example](#), [image classification](#).

Linearity measure: produces a score quantifying how linear the model is around a test instance. The linearity score measures the model linearity around a test instance by feeding the model linear superpositions of inputs and comparing the outputs with the linear combination of outputs from predictions on single inputs. [Documentation](#), [Tabular example](#), [image classification](#).

3.3 Prototypes

These algorithms provide a **distilled** view of the dataset and help construct a 1-KNN **interpretable** classifier.

Method	Classifica-tion	Regres-sion	Tabu-lar	Text	Im-ages	Categorical features	Fea-	Train set la-bels
ProtoSe-lect	✓		✓	✓	✓	✓		Optional

ProtoSelect: produces a condensed view of the training dataset and facilitates the construction of an interpretable classification model through 1-KNN. Every class k of the training dataset is summarised by a prototype set constructed to encourage the following three properties: i) covers as many training points as possible of the class k ; ii) covers as few training points as possible of classes different from k ; iii) is sparse - contains as few prototypes as possible. The method can be applied to any data modality as long as there is a meaningful way of defining a “distance” between data points which can often be done using a domain-specific pre-processing function. [Documentation](#), [Tabular and image example](#).

¹ depending on model

² may require dimensionality reduction

WHITE-BOX AND BLACK-BOX MODELS

Explainer algorithms can be categorised in many ways (see [this table](#)), but perhaps the most fundamental one is whether they work with **white-box** or **black-box** models.

White-box is a term used for any model that the explainer method can “look inside” and manipulate arbitrarily. In the context of `alibi` this category of models corresponds to Python objects that represent models, for example instances of `sklearn.base.BaseEstimator`, `tensorflow.keras.Model`, `torch.nn.Module` etc. The exact type of the white-box model in question enables different white-box explainer methods. For example, `tensorflow` and `torch` models are *gradient-based* which enables *gradient-based* explainer methods such as [Integrated Gradients](#)¹ whilst various types of tree-based models are supported by [TreeShap](#).

On the other hand, a **black-box** model describes any model that the explainer method may not inspect and modify arbitrarily. The only interaction with the model is via calling its `predict` function (or similar) on data and receiving predictions back. In the context of `alibi` black-box models have a concrete definition—they are functions that take in a `numpy` array representing data and return a `numpy` array representing a prediction. Using [type hints](#) we can define a general black-box model (also referred to as a *prediction function*) to be of type `Callable[[np.ndarray], np.ndarray]`². Explainers that expect black-box models as input are very flexible as *any* type of function that conforms to the expected type can be explained by black-box explainers.

Note: In addition to the expected type, black-box models **must** be compatible with batch prediction. I.e. `alibi` explainers assume that the first dimension of the input array is always batch.

Warning: There is currently one exception to the black-box interface: the `AnchorText` explainer expects the prediction function to be of type `Callable[[List[str], np.ndarray], np.ndarray]`, i.e. the model is expected to work on batches of raw text (here `List[str]` indicates a batch of text strings). See [this example](#) for more information.

4.1 Wrapping white-box models into black-box models

Models in Python all start out as white-box models (i.e. custom Python objects from some modelling library like `sklearn` or `tensorflow`). However, to be used with explainers that expect a black-box prediction function, the user has to define a prediction function that conforms to the black-box definition given above. Here we give a few common examples and some pointers about creating a black-box prediction function from a white-box model. In what follows we distinguish between the original white-box model and the wrapped black-box predictor function.

¹ At the time of writing `IntegratedGradients` only supports `tensorflow` models.

² Note that this definition limits black-box models to be *single-input* and *single-output* which are what most black-box `alibi` explainers can handle. In the general case the definition can be extended to *multi-input* and *multi-output* models, i.e. taking in and/or returning multiple arrays.

4.1.1 Scikit-learn models

All sklearn models expose a `predict` method that already conforms to the black-box function interface defined above which makes it easy to create black-box predictors:

```
predictor = model.predict
explainer = SomeExplainer(predictor, **kwargs)
```

In some cases for classifiers it may be more appropriate to expose the `predict_proba` or `decision_function` method instead of `predict`, see an example on [ALE for classifiers](#).

4.1.2 Tensorflow models

Tensorflow models (specifically instances of `tensorflow.keras.Model`) expose a `predict` method that takes in numpy arrays and returns predictions as numpy arrays³:

```
predictor = model.predict
explainer = SomeExplainer(predictor, **kwargs)
```

4.1.3 Pytorch models

Pytorch models (specifically instances of `torch.nn.Module`) expect and return instances of `torch.Tensor` from the `forward` method, thus we need to do a bit more work to define the predictor black-box function:

```
model.eval()

@torch.no_grad()
def predictor(X: np.ndarray) -> np.ndarray:
    X = torch.as_tensor(X, dtype=dtype, device=device)
    return model.forward(X).cpu().numpy()
```

Note that there are a few differences with tensorflow models:

- Ensure the model is in the evaluation mode (i.e., `model.eval()`) and that the mode does not change to training (i.e., `model.train()`) between consecutive calls to the explainer. Otherwise consider including `model.eval()` inside the predictor function.
- Decorate the predictor with `@torch.no_grad()` to avoid the computation and storage of the gradients which are not needed.
- Explicit conversion to a tensor with a specific `dtype`. Whilst tensorflow handles this internally when `predict` is called, for torch we need to do this manually.
- Explicit device selection for the tensor. This is an important step as numpy arrays are limited to cpu and if your model is on a gpu it will expect its input tensors to be on a gpu.
- Explicit conversion of prediction tensor to numpy. We first send the output to the cpu and then transform into numpy array.

If you are using [Pytorch Lightning](#) to create torch models, then the `dtype` and `device` can be retrieved as attributes of your `LightningModule`, see [here](#).

³ This is in contrast to the `__call__` and `call` methods which expect and return `tensorflow.Tensor` objects. However, using `__call__` may be preferable for performance in some cases (this would require transforming inputs and outputs similar to the torch example).

4.1.4 General models

Given the above examples, the pattern for defining a black-box predictor from a white-box model is clear: define a `predictor` function that manipulates the inputs to and outputs from the underlying model in a way that conforms to the black-box model interface `alibi` expects:

```
def predictor(X: np.ndarray) -> np.ndarray:
    inp = transform_input(X)
    output = model(inp) # or call the model-specific prediction method
    output = transform_output(output)
    return output

explainer = SomeExplainer(predictor, **kwargs)
```

Here `transform_input` and `transform_output` are general user-defined functions that appropriately transform the input `numpy` arrays in a format that the model expects and transform the output predictions into a `numpy` array so that `predictor` is an `alibi` compatible black-box function.

SAVING AND LOADING

Alibi includes experimental support for saving and loading a subset of explainers to disk using the `dill` module.

To save an explainer, simply call the `save` method and provide a path to a directory (a new one will be created if it doesn't exist):

```
explainer.save('path')
```

Alibi doesn't save the model/prediction function that is passed into the explainer so when loading the explainer you will need to provide it again:

```
from alibi.saving import load_explainer
explainer = load_explainer('path', predictor=predictor)
```

5.1 Details and limitations

Every explainer will save the following artifacts as a minimum:

```
path/meta.dill
path/explainer.dill
```

Here `meta.dill` is the metadata of the explainer (including the Alibi version used to create it) and `explainer.dill` is the serialized explainer. Some explainers may save more artifacts, e.g. `AnchorText` additionally saves `path/nlp` which is the `spacy` model used to initialize the explainer using the native `spacy` saving functionality (pickle based) whilst `AnchorImage` also saves the custom Python segmentation function under `path/segmentation_fn.dill`.

When loading a saved explainer, a warning will be issued if the runtime Alibi version is different from the version used to save the explainer. **It is highly recommended to use the same Alibi, Python and dependency versions as were used to save the explainer to avoid potential bugs and incompatibilities.**

FREQUENTLY ASKED QUESTIONS

6.1 General troubleshooting

6.1.1 I'm getting code errors using a method on my model and my data

There can be many reasons why the method does not work. For code exceptions it is a good idea to check the following:

- Read the *docstrings* of the method, paying close attention to the *type hints* as most errors are due to misformatted input arguments.
- Check that your model signature (its type and expected inputs/outputs) are in the right format. Typically this means taking as input a `numpy` array representing a batch of data and returning a `numpy` array representing class labels, probabilities or regression values. For further details refer to *White-box and black-box models*.
- Check the expected input type for the `explain` method. Note that in many cases (e.g. all the `Anchor` methods) the `explain` method expects a single instance *without* a leading batch dimension, e.g. for `AnchorImage` a colour image of shape (height, width, colour_channels).

6.1.2 My model works on different input types, e.g. pandas dataframes instead of numpy arrays so the explainers don't work

At the time of writing we support models that operate on `numpy` arrays. You can write simple wrapper functions for your model so that it adheres to the format that `alibi` expects, *see here*. In the future we may support more diverse input types (see #516).

6.1.3 Explanations are taking a long time to complete

Explanations can take different times as a function of the model, the data, and the explanation type itself. Refer to the *Introduction* and *Methods* sections for notes on algorithm complexity. You might need to experiment with the type of model, data points (specifically feature cardinality), and method parameters to ascertain if a specific method scales well for your use case.

6.1.4 The explanation returned doesn't make sense to me

Explanations reflect the decision-making process of the model and not that of a biased human observer, see [Biases](#). Moreover, depending on the method, the data, and the model, the explanations returned are valid but may be harder to interpret (e.g. see [Anchor explanations](#) for some examples).

6.1.5 Is there a way I can get more information from the library during the explanation generation process?

Yes! We use [Python logging](#) to log info and debug messages from the library. You can configure logging for your script to see these messages during the execution of the code. Additionally, some methods also expose a `verbose` argument to print information to standard output. Configuring Python logging for your application will depend on your needs, but for simple scripts you can easily configure logging to print to standard error as follows:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Note: this will display *all* logged messages with level `DEBUG` and higher from *all* libraries in use.

6.2 Anchor explanations

6.2.1 Why is my anchor explanation empty (tabular or text data) or black (image data)?

This is expected behaviour and signals that there is no salient subset of features that is necessary for the prediction to hold. In other words, with high probability (as measured by the precision), the predicted class of the data point does not change regardless of the perturbations applied to it.

Note: this behaviour can be typical for very imbalanced datasets, [see comment from the author](#).

6.2.2 Why is my anchor explanation so long (tabular or text data) or covers much of the image (image data)?

This is expected behaviour and can happen in two ways:

- The data point to be explained lies near the decision boundary of the classifier. Thus, many more predicates are needed to ensure that a data point keeps the predicted class as small changes to the feature values may push the prediction to another class.
- For tabular data, sampling perturbations is done using a training set. If the training set is imbalanced, explaining a minority class data point will result in oversampling perturbed features typical of majority classes so the algorithm would struggle to find a short anchor exceeding the specified precision level. For a concrete example, see [Anchor explanations for income prediction](#).

6.3 Counterfactual explanations

6.3.1 I'm using the methods Counterfactual, CounterfactualProto, or CEM on a tree-based model such as decision trees, random forests, or gradient boosted models (e.g. xgboost) but not finding any counterfactual examples

These methods only work on a subset of black-box models, namely ones whose decision function is differentiable with respect to the input data and hence amenable to gradient-based counterfactual search. Since for tree-based models, the decision function is piece-wise constant these methods won't work. It is recommended to use *CFRL* instead.

6.3.2 I'm getting an error using the methods Counterfactual, CounterfactualProto, or CEM, especially if trying to use one of these methods together with IntegratedGradients or CFRL

At the moment the 3 counterfactual methods are implemented using TensorFlow 1.x constructs. This means that when running these methods, first we need to disable behaviour specific to TensorFlow 2.x as follows:

```
import tensorflow as tf
tf.compat.v1.disable_v2_behavior()
```

Unfortunately, running this line means it's impossible to run explainers based on TensorFlow 2.x such as *IntegratedGradients* or *CFRL*. Thus at the moment, it is impossible to run these explainers together in the same Python interpreter session. Ultimately the fix is to rewrite the TensorFlow 1.x implementations in idiomatic TensorFlow 2.x and [some work has been done](#) but is currently not prioritised.

6.3.3 Why am I'm unable to restrict the features allowed to changed in CounterfactualProto?

This is a known issue with the current implementation, see [here](#) and [here](#). It is currently blocked until we migrate the code to use TensorFlow 2.x constructs. In the meantime, it is recommended to use *CFRL* for counterfactual explanations with flexible feature-range constraints.

6.4 Similarity explanations

6.4.1 I'm using the GradientSimilarity method on a large model and it runs very slow. If I use precompute_grads=True I get out of memory errors. How do I solve this?

Large models with many parameters result in the similarity method running very slow and using `precompute_grads=True` may not be an option due to the memory cost. The best solutions for this problem are:

- Use the explainer on a reduced dataset. You can use *Prototype Methods* to obtain a smaller representative sample.
- Freeze some parameters in the model so that when computing the gradients the similarity method excludes them. If using `tensorflow` you can do this by setting `trainable=False` on layers or specific parameters. For `pytorch` we can set `requires_grad=False` on the relevant model parameters.

Note that doing so will cause the explainer to issue a warning on initialization, informing you there are non-trainable parameters in your model and the explainer will not use those when computing the similarity scores.

6.4.2 I'm using the GradientSimilarity method on a tensorflow model and I keep getting warnings about non-trainable parameters but I haven't set any to be non-trainable?

This warning likely means that your model has layers that track statistics using non-trainable parameters such as batch normalization layers. The warning should list the specific tensors that are non-trainable so you should be able to check. If this is the case you don't need to worry as similarity methods don't use those parameters anyway. Otherwise you will see this warning if you have set one of the parameters to `trainable=False` and alibi is just making sure you know this is the case.

[source]

7.1 Accumulated Local Effects

7.1.1 Overview

Accumulated Local Effects (ALE) is a method for computing feature effects based on the paper [Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models](#) by Apley and Zhu. The algorithm provides model-agnostic (*black box*) global explanations for classification and regression models on tabular data.

ALE addresses some key shortcomings of [Partial Dependence Plots](#) (PDP), a popular method for estimating first order feature effects. We discuss these limitations and motivate ALE after presenting the method usage.

7.1.2 Usage

Initialize the explainer by passing a black-box prediction function and optionally a list of feature names and target (class) names for interpretation:

```
from alibi.explainers import ALE
ale = ALE(predict_fn, feature_names=feature_names, target_names=target_names)
```

Following the initialization, we can immediately produce an explanation given a dataset of instances X :

```
exp = ale.explain(X)
```

The `explain` method has a default argument, `min_bin_points=4`, which determines the number of bins the range of each feature is subdivided into so that the ALE estimate for each bin is made with at least `min_bin_points`. Smaller values can result in less accurate local estimates while larger values can also result in less accurate estimates by averaging across large parts of the feature range.

Alternatively, we can run the explanation only on a subset of features:

```
exp = ale.explain(X, features=[0, 1])
```

This is useful if the number of total features is large and only small number is of interest. Also, it can be particularly useful to filter out categorical variable columns as there is no consistent ALE formulation and hence any results for categorical variables would be misleading.

It is also possible to define custom grid points for each feature. The custom grid points are defined in a dictionary where the keys are the features indices and the values are numpy arrays containing the points. The sequence of points for each feature must be monotonically increasing:

```
grid_points = {0: np.array([0.1, 0.5, 1.0]),
               1: np.array([0.1, 0.5, 1.0])}
exp = ale.explain(X, features=[0, 1], grid_points=grid_points)
```

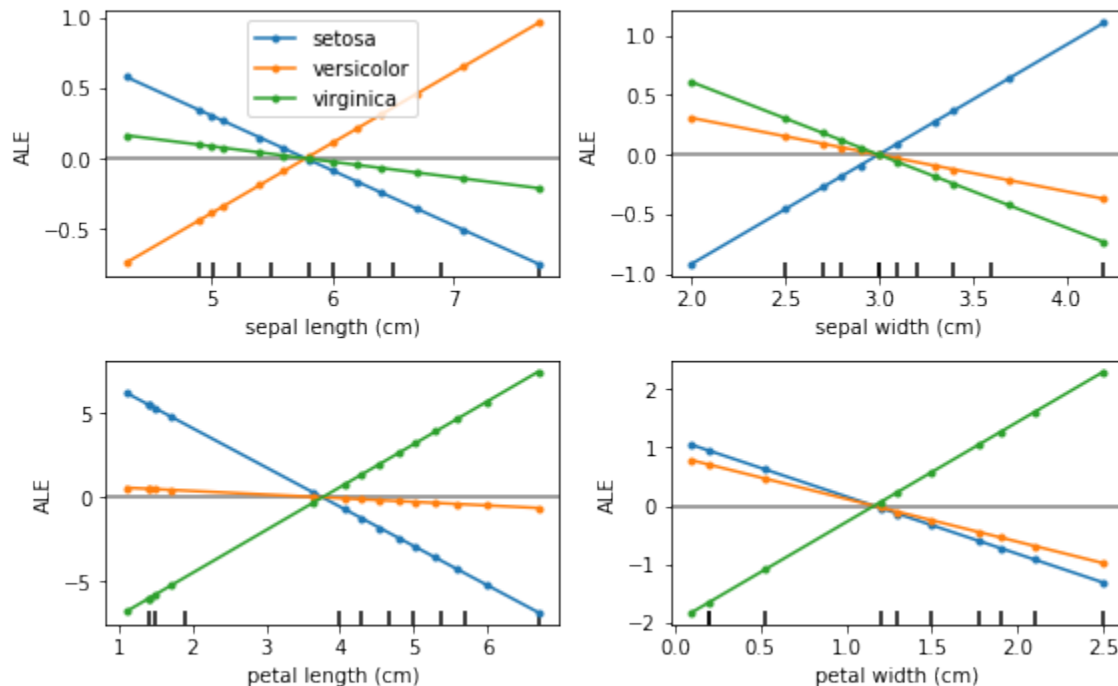
The result `exp` is an Explanation object which contains the following data-related attributes:

- `ale_values` - a list of arrays of ALE values (one for each feature). Each array can have multiple columns (if the number of targets is >1 as in classification).
- `constant_value` - the mean prediction over X (zeroth order effects).
- `ale0` - a list of arrays of “centering” values (one for each feature) used by the algorithm to center the `ale_values` around the expected effect for the feature (i.e. the sum of `ale_values` and `ale0` will be the uncentered ALE).
- `feature_values` - a list of arrays (one for each feature) of feature values at which the ALE values were computed.
- `feature_names` - an array of feature names.
- `target_names` - an array of target names.
- `feature_deciles` - a list of arrays (one for each feature) of the feature deciles.

Plotting `ale_values` against `feature_values` recovers the ALE curves. For convenience we include a plotting function `plot_ale` which automatically produces ALE plots using `matplotlib`:

```
from alibi.explainers import plot_ale
plot_ale(exp)
```

The following is an example ALE plot of a logistic regression model on the Iris dataset (see worked [example](#)):



7.1.3 Examples

ALE regression example (California house prices)

ALE classification example (Iris dataset)

7.1.4 Motivation and definition

The following exposition largely follows [Apley and Zhu \(2016\)](#) and [Molnar \(2019\)](#).

Given a predictive model $f(x)$ where $x = (x_1, \dots, x_d)$ is a vector of d features, we are interested in computing the *feature effects* of each feature x_i on the model $f(x)$. A feature effect of feature x_i is some function $g(x_i)$ designed to disentangle the contribution of x_i to the response $f(x)$. To simplify notation, in the following we consider the $d = 2$ case and define the feature effect functions for the first feature x_1 .

Partial Dependence

Partial Dependence Plots (PDP) is a very common method for computing feature effects. It is defined as

$$\text{PD}(x_1) = \mathbb{E}[f(x_1, X_2)] = \int p(x_2) f(x_1, x_2) dx_2,$$

where $p(x_2)$ is the marginal distribution of X_2 . To estimate the expectation, we can take the training set X and average the predictions of instances where the first feature for all instances is replaced by x_1 :

$$\widehat{\text{PD}}(x_1) = \frac{1}{n} \sum_{j=1}^n f(x_1, x_{2,j}).$$

The PD function attempts to calculate the effect of x_1 by averaging the effects of the other feature x_2 over its marginal distribution. This is problematic because by doing so we are averaging predictions of many *out of distribution* instances. For example, if x_1 and x_2 are a person's height and weight and f predicts some other attribute of the person, then the PD function at a fixed height x_1 would average predictions of persons with height x_1 and *all possible weights* x_2 observed in the training set. Clearly, since height and weight are strongly correlated this would lead to many unrealistic data points. Since the predictor f has not been trained on such impossible data points, the predictions are no longer meaningful. We can say that an implicit assumption motivating the PD approach is that the features are uncorrelated, however this is rarely the case and severely limits the usage of PDP.

An attempt to fix the issue with the PD function is to average over the conditional distribution instead of the marginal which leads to the following feature effect function:

$$M(x_1) = \mathbb{E}[f(X_1, X_2) | X_1 = x_1] = \int p(x_2 | x_1) f(x_1, x_2) dx_2,$$

where $p(x_2 | x_1)$ is the conditional distribution of X_2 . To estimate this function from the training set X we can compute

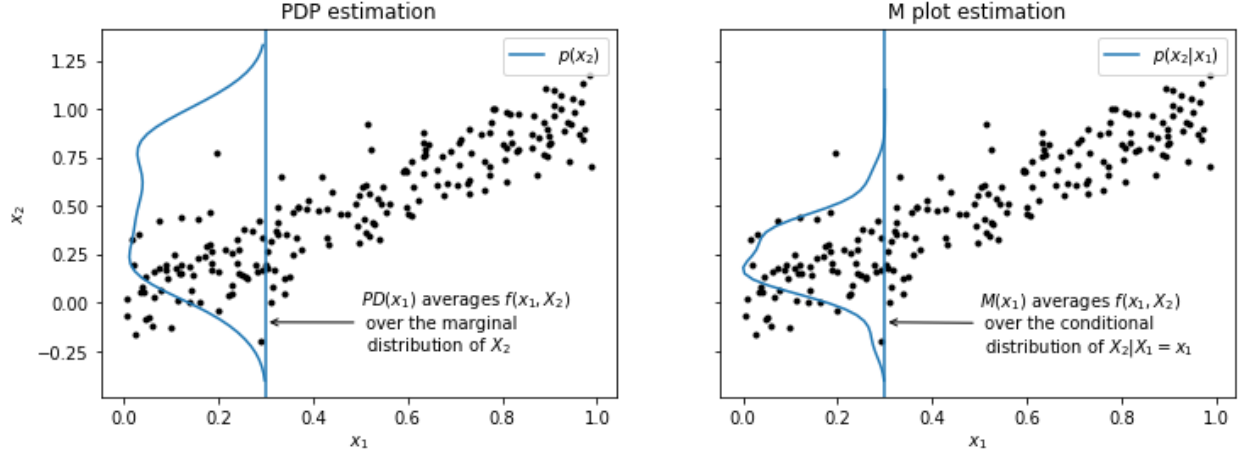
$$\widehat{M}(x_1) = \frac{1}{n(x_1)} \sum_{j \in N(x_1)} f(x_1, x_{2,j}),$$

where $N(x_1)$ is a subset of indices j for which $x_{1,j}$ falls into some small neighbourhood of x_1 and $n(x_1)$ is the number of such instances.

While this refinement addresses the issue of the PD function averaging over impossible data points, the use of the $M(x_1)$ function as feature effects remains limited when the features are correlated. To go back to the example with people's height and weight, if we fix the height to be some particular value x_1 and calculate the effects according to $M(x_1)$, because of the correlation of height and weight the function value mixes effects of *both* features and estimates the **combined** effect. This is undesirable as we cannot attribute the value of $M(x_1)$ purely to height. Furthermore,

suppose height doesn't actually have any effect on the prediction, only weight does. Because of the correlation between height and weight, $M(x_1)$ would still show an effect which can be highly misleading. Concretely, for a model like $f(x_1, x_2) = x_2$ it is possible that $M(x_1) \neq 0$ if x_1, x_2 are correlated.

The following plot summarizes the two approaches for estimating the effect of x_1 at a particular value when x_2 is strongly correlated with x_1 :



ALE

ALE solves the problem of mixing effects from different features. As with the function $M(x_1)$, ALE uses the conditional distribution to average over other features, but instead of averaging the predictions directly, it averages *differences in predictions* to block the effect of correlated features. The ALE function is defined as follows:

$$\text{ALE}(x_1) = \int_{\min(x_1)}^{x_1} \mathbb{E} \left[\frac{\partial f(X_1, X_2)}{\partial X_1} \middle| X_1 = z_1 \right] dz_1 - c_1 \quad (7.1)$$

$$= \underbrace{\int_{\min(x_1)}^{x_1} \int p(x_2|z_1) \frac{\partial f(z_1, x_2)}{\partial z_1} dx_2 dz_1}_{\text{uncentered ALE}} - c_1, \quad (7.2)$$

where the constant c_1 is chosen such that the resulting ALE values are independent of the point $\min(x_1)$ and have zero mean over the distribution $p(x_1)$.

The term $\frac{\partial f(x_1, x_2)}{\partial x_1}$ is called the *local effect* of x_1 on f . Averaging the local effect over the conditional distribution $p(x_2|x_1)$ allows us to isolate the effect of x_1 from the effects of other correlated features avoiding the issue of M plots which directly average the predictor f . Finally, note that the local effects are integrated over the range of x_1 , this corresponds to the *accumulated* in ALE. This is done as a means of visualizing the *global* effect of the feature by “piecing together” the calculated local effects.

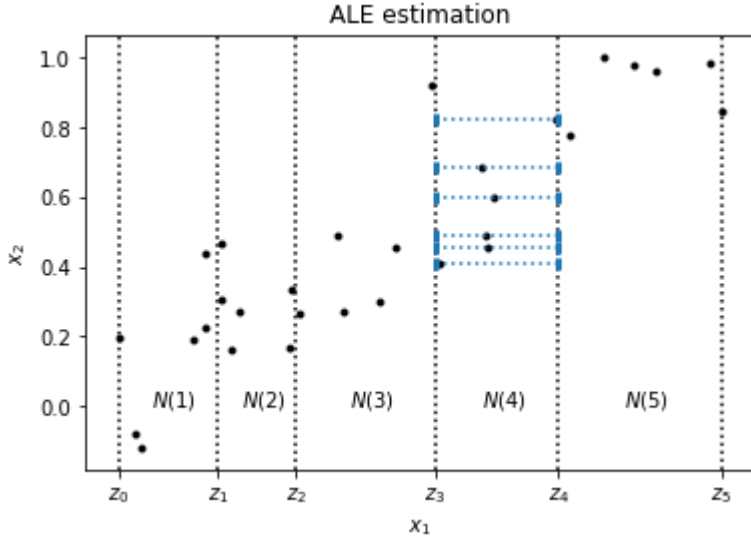
In practice, we calculate the local effects by finite differences so the predictor f need not be differentiable. Thus, to estimate the ALE from data, we compute the following:

$$\widehat{\text{ALE}}(x_1) = \underbrace{\sum_{k=1}^{k(x_1)} \frac{1}{n(k)} \sum_{i: x_1^{(i)} \in N(k)} \left[f(z_k, x_{\setminus 1}^{(i)}) - f(z_{k-1}, x_{\setminus 1}^{(i)}) \right]}_{\text{uncentered ALE}} - c_1.$$

Here z_0, z_1, \dots is a sufficiently fine grid of the feature x_1 (typically quantiles so that each resulting interval contains a similar number of points), $N(k)$ denotes the interval $[z_{k-1}, z_k)$, $n(k)$ denotes the number of points falling into interval

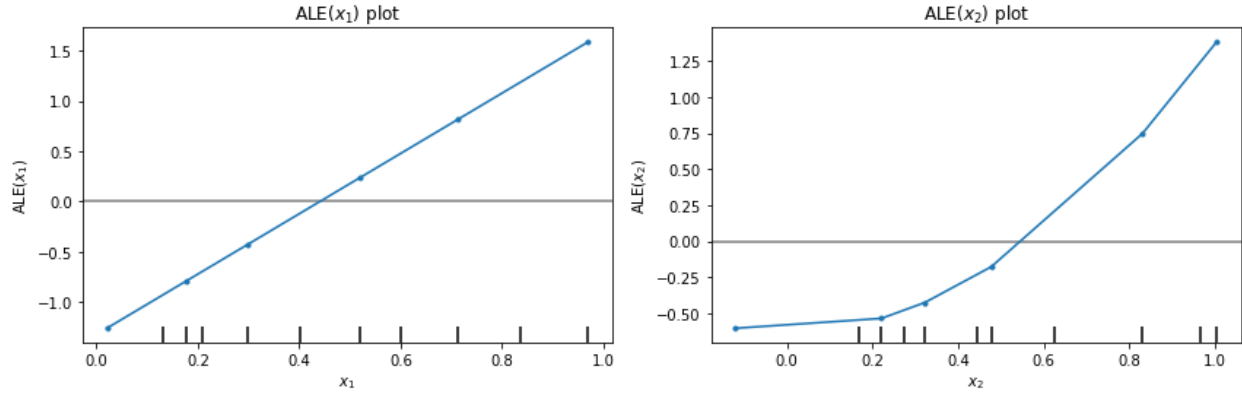
$N(k)$ and $k(x_1)$ denotes the index of the interval into which x_1 falls into, i.e. $x_1 \in [z_{k(x_1)-1}, z_{k(x_1)})$. Finally, the notation $f(z_k, x_{\setminus 1}^{(i)})$ means that for instance i we replace x_1 with the value of the right interval end-point z_k (likewise for the left interval end-point using z_{k-1}), leaving the rest of the features unchanged, and evaluate the difference of predictions at these points.

The following plot illustrates the ALE estimation process. We have subdivided the feature range of x_1 into 5 bins with roughly the same number of points indexed by $N(k)$. Focusing on bin $N(4)$, for each point falling into this bin, we replace their x_1 feature value by the left and right end-points of the interval, z_3 and z_4 . Then we evaluate the difference of the predictions of these points and calculate the average by dividing by the number of points in this interval $n(4)$. We do this for every interval and sum up (accumulate) the results. Finally, to calculate the constant c_1 , we subtract the expectation over $p(x_1)$ of the calculated uncentered ALE so that the resulting ALE values have mean zero over the distribution $p(x_1)$.



We show the results of ALE calculation for a model $f(x_1, x_2) = 3x_1 + 2x_2^2$. The resulting plots correctly recover the linear effect of x_1 and the quadratic effect of x_2 on f . Note that the ALE is estimated for each interval edge and linearly interpolated in between, for real applications it is important to have a sufficiently fine grid but also one that has enough points into each interval for accurate estimates. The x-axis also shows feature deciles of the feature to help judge in which parts of the feature space the ALE plot is interpolating more and the estimate might be less trustworthy.

The value of $ALE(x_i)$ is the main effect of feature x_i as compared to the average effect of that feature. For example, the value of $ALE(x_1) = 0.75$ at $x_1 = 0.7$, if we sample data from the joint distribution $p(x_1, x_2)$ (i.e. realistic data points) and $x_1 = 0.7$, then we would expect the first order effect of feature x_1 to be 0.75 higher than the *average* first order effect of this feature. Seeing that the $ALE(x_1)$ plot crosses zero at $x_1 \approx 0.45$, realistic data points with $x_1 \approx 0.45$ will have effect on f similar to the average first order effect of x_1 . For realistic data points with smaller x_1 , the effect will become negative with respect to the average effect.



Note

The interpretation of ALE plots is mainly qualitative—the focus should be on the shape of the curve at different points in the feature range. The steepness of the tangent to the curve (or the slope of the linear interpolation) determines the size of the effect of that feature locally—the steeper the tangent, the bigger the effect.

Because the model $f(x_1, x_2) = 3x_1 + 2x_2^2$ is explicit and differentiable, we can calculate the ALE functions analytically which gives us even more insight. The partial derivatives are given by $(3, 4x_2)$. Assuming that the conditional distributions $p(x_2|x_1)$ and $p(x_1|x_2)$ are uniform, the expectations over the conditional distributions are equal to the partial derivatives. Next, we integrate over the range of the features to obtain the *uncentered* ALE functions:

$$\text{ALE}_u(x_1) = \int_{\min(x_1)}^{x_1} 3dz_1 = 3x_1 - 3\min(x_1) \quad (7.3)$$

$$\text{ALE}_u(x_2) = \int_{\min(x_2)}^{x_2} 4z_2dz_2 = 2x_2^2 - 2\min(x_2)^2. \quad (7.4)$$

Finally, to obtain the ALE functions, we center by setting $c_i = \mathbb{E}(\text{ALE}_u(x_i))$ where the expectation is over the marginal distribution $p(x_i)$:

$$\text{ALE}(x_1) = 3x_1 - 3\min(x_1) - \mathbb{E}(3x_1 - 3\min(x_1)) = 3x_1 - 3\mathbb{E}(x_1) \quad (7.5)$$

$$\text{ALE}(x_2) = 2x_2^2 - 2\min(x_2)^2 - \mathbb{E}(2x_2^2 - 2\min(x_2)^2) = 2x_2^2 - 2\mathbb{E}(x_2^2). \quad (7.6)$$

This calculation verifies that the ALE curves are the desired feature effects (linear for x_1 and quadratic for x_2) relative to the mean feature effects across the dataset. In fact if f is additive in the individual features like our toy model, then the ALE main effects recover the correct additive components (Apley and Zhu (2016)). Furthermore, for additive models we have the decomposition $f(x) = \mathbb{E}(f(x)) + \sum_{i=1}^d \text{ALE}(x_i)$, here the first term which is the average prediction across the dataset X can be thought of as zeroth order effects.

[source]

7.2 Anchors

7.2.1 Overview

The anchor algorithm is based on the [Anchors: High-Precision Model-Agnostic Explanations](#) paper by Ribeiro et al.(2018) and builds on the open source [code](#) from the paper's first author.

The algorithm provides model-agnostic (*black box*) and human interpretable explanations suitable for classification models applied to images, text and tabular data. The idea behind anchors is to explain the behaviour of complex models with high-precision rules called *anchors*. These anchors are locally sufficient conditions to ensure a certain prediction with a high degree of confidence.

Anchors address a key shortcoming of local explanation methods like [LIME](#) which proxy the local behaviour of the model in a linear way. It is however unclear to what extent the explanation holds up in the region around the instance to be explained, since both the model and data can exhibit non-linear behaviour in the neighborhood of the instance. This approach can easily lead to overconfidence in the explanation and misleading conclusions on unseen but similar instances. The anchor algorithm tackles this issue by incorporating coverage, the region where the explanation applies, into the optimization problem. A simple example from sentiment classification illustrates this (Figure 1). Dependent on the sentence, the occurrence of the word *not* is interpreted as positive or negative for the sentiment by LIME. It is clear that the explanation using *not* is very local. Anchors however aim to maximize the coverage, and require *not* to occur together with *good* or *bad* to ensure respectively negative or positive sentiment.

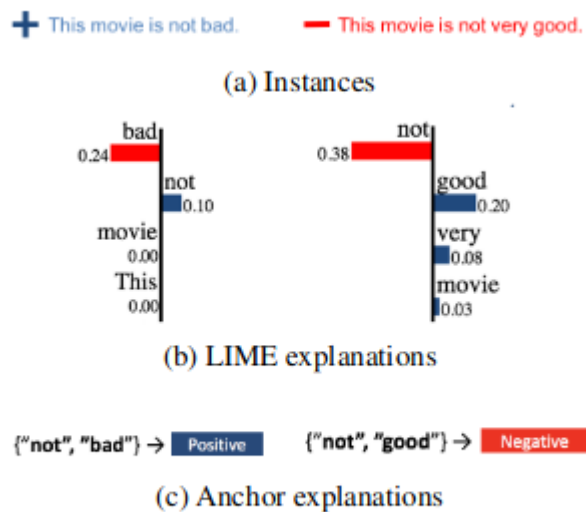


Figure 1: Sentiment predictions, LSTM

Ribeiro et al., *Anchors: High-Precision Model-Agnostic Explanations*, 2018

As highlighted by the above example, an anchor explanation consists of *if-then rules*, called the anchors, which sufficiently guarantee the explanation locally and try to maximize the area for which the explanation holds. This means that as long as the anchor holds, the prediction should remain the same regardless of the values of the features not present in the anchor. Going back to the sentiment example: as long as *not good* is present, the sentiment is negative, regardless of the other words in the movie review.

7.2.2 Concepts and use-case insights

For a more intuitive understanding of what the method tries to achieve, we will loosely define a few concepts and explain some insights we get from an anchor explanation.

A **predicate** represents an expression involving a single feature. Some examples of predicates for a tabular dataset having features such as *Age*, *Relationship*, and *Occupation* are:

- `28 < Age < 50`
- `Relationship = Husband`
- `Occupation = Blue-Collar`

A **rule** represents a set of predicates connected by the AND operator. Considering all the predicate examples above, we can construct the following rule: `28 < Age < 50 AND Relationship = Husband AND Occupation = Blue-Collar`. Note that a rule selects/refers to a particular subpopulation from the given dataset.

We can now define the notion of an **anchor**. Following the definition from [Ribeiro et al. \(2018\)](#), “an **anchor** explanation is a **rule** that sufficiently ‘anchors’ the prediction locally – such that changes to the rest of the feature values of the instance do not matter”.

As previously mentioned, the power of the Anchors over other local explanations methods comes from the objective formulation which is to maximize the **coverage** under the **precision** constraints.

Precision represents the probability of receiving the same classification label of the explained input if we query the model on other instances that satisfy the anchor predicates. The expected precision range is the interval $[t, 1]$, where t is the user-specified precision threshold.

Coverage represents the proportion of the population which satisfy the anchor predicates. It is a positive number ≤ 1 , where a value of 1 corresponds to the empty anchor.

There are some edge cases that a practitioner should be aware of:

- An anchor with many predicates and a small coverage might indicate that the explained input lies near the decision boundary. Many more predicates are needed to ensure that an instance keeps the predicted label since minor perturbations may push the prediction to another class.
- An empty anchor with a coverage of 1 indicates that there is no salient subset of features that is necessary for the prediction to hold. In other words, with high probability (as measured by the precision), the predicted class of the data point does not change regardless of the perturbations applied to it. This behaviour can be typical for very imbalanced datasets.

Check [FAQ](#) for further details.

7.2.3 Data modalities

Text

For text classification, an interpretable anchor consists of the words that need to be present to ensure a prediction, regardless of the other words in the input. The words that are not present in a candidate anchor can be sampled in 3 ways:

- Replace word token by UNK token.
- Replace word token by sampled token from a corpus with the same POS tag and probability proportional to the similarity in the embedding space. By sampling similar words, we keep more context than simply using the UNK token.

- Replace word tokens with sampled tokens according to the masked language model probability distribution. The tokens can be sampled in parallel, independent of one another, or sequentially (autoregressive), conditioned on the previously generated tokens.

Tabular Data

anchors are also suitable for tabular data with both categorical and continuous features. The continuous features are discretized into quantiles (e.g. deciles), so they become more interpretable. The features in a candidate anchor are kept constant (same category or bin for discretized features) while we sample the other features from a training set. As a result, anchors for tabular data need access to training data. Let's illustrate this with an example. Say we want to predict whether a person makes less or more than £50,000 per year based on the person's characteristics including age (continuous variable) and marital status (categorical variable). The following would then be a potential anchor: Hugo makes more than £50,000 because he is married and his age is between 35 and 45 years.

Images

Similar to LIME, images are first segmented into superpixels, maintaining local image structure. The interpretable representation then consists of the presence or absence of each superpixel in the anchor. It is crucial to generate meaningful superpixels in order to arrive at interpretable explanations. The algorithm supports a number of standard image segmentation algorithms ([felzenszwalb](#), [slic](#) and [quickshift](#)) and allows the user to provide a custom segmentation function.

The superpixels not present in a candidate anchor can be masked in 2 ways:

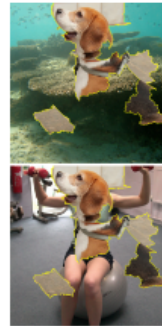
- Take the average value of that superpixel.
- Use the pixel values of a superimposed picture over the masked superpixels.



(a) Original image



(b) Anchor for "beagle"



(c) Images where Inception predicts $P(\text{beagle}) > 90\%$

Ribeiro et al., *Anchors: High-Precision Model-Agnostic Explanations*, 2018

Efficiently Computing Anchors

The anchor needs to return the same prediction as the original instance with a minimal confidence of e.g. 95%. If multiple candidate anchors satisfy this constraint, we go with the anchor that has the largest coverage. Because the number of potential anchors is exponential in the feature space, we need a faster approximate solution.

The anchors are constructed bottom-up in combination with [beam search](#). We start with an empty rule or anchor, and incrementally add an *if-then* rule in each iteration until the minimal confidence constraint is satisfied. If multiple valid anchors are found, the one with the largest coverage is returned.

In order to select the best candidate anchors for the beam width efficiently during each iteration, we formulate the problem as a [pure exploration multi-armed bandit](#) problem. This limits the number of model prediction calls which can be a computational bottleneck.

For more details, we refer the reader to the original [paper](#).

7.2.4 Usage

While each data type has specific requirements to initialize the explainer and return explanations, the underlying algorithm to construct the anchors is the same.

In order to efficiently generate anchors, the following hyperparameters need to be set to sensible values when calling the `explain` method:

- **threshold**: Minimum anchor precision threshold. The algorithm tries to find an anchor that maximizes the coverage under precision constraint. The precision constraint is formally defined as $P(\text{prec}(A) \geq t) \geq 1 - \delta$, where A is an anchor, t is the **threshold** parameter, δ is the **delta** parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, we are seeking for an anchor having its precision greater or equal than the given **threshold** with a confidence of $(1 - \text{delta})$. A higher value guarantees that the anchors are faithful to the model, but also leads to more computation time. Note that there are cases in which the precision constraint cannot be satisfied due to the quantile-based discretisation of the numerical features. If that is the case, the best (i.e. highest coverage) non-eligible anchor is returned. The default value is 0.95.
- **delta**: Significance threshold. $1 - \text{delta}$ represents the confidence threshold for the anchor precision (see **threshold**) and the selection of the best anchor candidate in each iteration (see **tau**).
- **tau**: Multi-armed bandit parameter used to select candidate anchors in each iteration. The multi-armed bandit algorithm tries to find within a tolerance **tau** the most promising (i.e. according to the precision) **beam_size** candidate anchor(s) from a list of proposed anchors. Formally, when the **beam_size**=1, the multi-armed bandit algorithm seeks to find an anchor A such that $P(\text{prec}(A) \geq \text{prec}(A^*) - \tau) \geq 1 - \delta$, where A^* is the anchor with the highest true precision (which we don't know), τ is the **tau** parameter, δ is the **delta** parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, in each iteration, the algorithm returns with a probability of at least $1 - \text{delta}$ an anchor A with a precision within an error tolerance of **tau** from the precision of the highest true precision anchor A^* . A bigger value for **tau** means faster convergence but also looser anchor conditions.
- **batch_size**: Batch size used for sampling. The Anchor algorithm will query the black-box model in batches of size **batch_size**. A larger **batch_size** gives more confidence in the anchor, again at the expense of computation time since it involves more model prediction calls. The default value is 100.
- **coverage_samples**: Number of samples used to estimate coverage from during result search. By default set to 10000.
- **beam_size**: Number of candidate anchors selected by the multi-armed bandit algorithm in each iteration from a list of proposed anchors. A bigger beam width can lead to a better overall anchor (i.e. prevents the algorithm of getting stuck in a local maximum) at the expense of more computation time.

Text

[\[source\]](#)

Predictor

Since the explainer works on black-box models, only access to a predict function is needed. The model below is a simple logistic regression trained on movie reviews with negative or positive sentiment and pre-processed with a CountVectorizer:

```
predict_fn = lambda x: clf.predict(vectorizer.transform(x))
```

Simple sampling strategies

AnchorText provides two simple sampling strategies: `unknown` and `similarity`. Randomly chosen words, except those in queried anchor, are replaced by the UNK token for the `unknown` strategy, and by similar words with the same part of speech of tag for the `similarity` strategy.

To perform text tokenization, pos-tagging, compute word similarity, etc., we use spaCy. The spaCy model can be loaded as follows:

```
import spacy
from alibi.utils import spacy_model

model = 'en_core_web_md'
spacy_model(model=model)
nlp = spacy.load(model)
```

If we choose to replace words with the UNK token, we define the explainer as follows:

```
explainer = AnchorText(predictor=predict_fn, sampling_strategy='unknown', nlp=nlp)
```

Likewise, if we choose to sample similar words from a corpus, we define the explainer as follows:

```
explainer = AnchorText(predictor=predict_fn, sampling_strategy='similarity', nlp=nlp)
```

Language model

AnchorText provides the option to define the perturbation distribution through a `language_model` sampling strategy. In this case, randomly chosen words, except those in the queried anchor, are replaced by words sampled according to the language model's predictions. We provide support for three transformer based language models: `DistilbertBaseUncased`, `BertBaseUncased`, and `RobertaBase`.

A language model can be loaded as follows:

```
language_model = DistilbertBaseUncased()
```

Then we can initialize the explainer as follows:

```
explainer = AnchorText(predictor=predict_fn, sampling_strategy="language_model",
                      language_model=language_model)
```

Sampling parameters

Parameters specific to each sampling strategy can be passed to the constructor via `kwargs`. For example:

- If `sampling_strategy="unknown"` we can initialize the explainer as follows:

```
explainer = AnchorText(
    predictor=predict_fn,
    sampling_strategy='unknown',      # replace a word by UNK token
    nlp=nlp,                        # spacy object
    sample_proba=0.5,                # probability of a word to be replaced by UNK
    ↪ token
)
```

- If `sampling_strategy="similarity"` we can initialize the explainer as follows:

```
explainer = AnchorText(
    predictor=predict_fn,
    sampling_strategy='similarity',   # replace a word by similar words
    nlp=nlp,                        # spacy object
    sample_proba=0.5,                # probability of a word to be replaced by as
    ↪ similar word
    use_proba=True,                  # sample according to the similarity distribution
    top_n=20,                        # consider only top 20 most similar words
    temperature=0.2                  # higher temperature implies more randomness when
    ↪ sampling
)
```

- Or if `sampling_strategy="language_model"`, the explainer can be defined as:

```
explainer = AnchorText(
    predictor=predict_fn,
    sampling_strategy="language_model", # use language model to predict the masked
    ↪ words
    language_model=language_model,     # language model to be used
    filling="parallel",                 # just one pass through the transformer
    sample_proba=0.5,                  # probability of masking and replacing a word
    ↪ according to the LM
    frac_mask_templates=0.1,            # fraction of masking templates
    use_proba=True,                    # use words distribution when sampling (if
    ↪ false sample uniform)
    top_n=50,                          # consider the fist 50 most likely words
    temperature=0.2,                   # higher temperature implies more randomness
    ↪ when sampling
    stopwords=['and', 'a', 'but'],      # those words will not be masked/disturbed
    punctuation=string.punctuation,    # punctuation tokens contained here will not
    ↪ be masked/disturbed
    sample_punctuation=False,           # if False tokens included in `punctuation`
    ↪ will not be sampled
    batch_size_lm=32                    # batch size used for the language model
)
```

Words outside of the candidate anchor can be replaced by UNK token, similar words, or masked out and replaced by the most likely words according to language model prediction, with a probability equal to `sample_proba`. We can sample the *top n* most similar words or the *top n* most likely language model predictions by setting the `top_n`

parameter. We can put more weight on similar or most likely words by decreasing the `temperature` argument. It is also possible to sample words from the corpus proportional to the word similarity with the ground truth word or according to the language model's conditional probability distribution by setting `use_proba` to `True`. Furthermore, we can avoid masking specific words by including them in the `stopwords` list.

Working with transformers can be computationally and memory-wise expensive. For `sampling_strategy="language_model"` we provide two methods to predict the masked words: `filling="parallel"` and `filling="autoregressive"`.

If `filling="parallel"`, we perform a single forward pass through the transformer. After obtaining the probability distribution of the masked words, each word is sampled independently of the others.

If `filling="autoregressive"`, we perform multiple forward passes through the transformer and generate the words one at a time. Thus, the masked words will be conditioned on the previous ones. **Note that this filling method is computationally expensive.**

To further decrease the explanation runtime, for `sampling_strategy="language_model"`, `filling="parallel"`, we provide a secondary functionality through the `frac_mask_templates`. Behind the scenes, the anchor algorithm is constantly requesting samples to query the predictor. Thus, we need to generate what we call *mask templates*, which are sentences containing words outside the candidate anchors replaced by the `<MASK>` token. The `frac_mask_templates` controls the fraction of mask templates to be generated. For example, if we need to generate 100 samples and the `frac_mask_templates=0.1`, we will generate only 10 mask templates. Those 10 templates are then passed to the language model to predict the masked words. Having the distribution of each word in each mask template, we can generate 100 samples as requested. Note that instead of passing 100 masked sentences through the language model (which is expensive), we only pass 10 sentences. Although this can increase the speed considerably, it can also decrease the diversity of the samples. The maximum batch size used in a forward pass through the language model can be specified by setting `batch_size_lm`.

When `sampling_strategy="language_model"`, we can specify the punctuation considered by the sampling algorithm. Any token composed only from characters in the `punctuation` string, will not be perturbed (we call those *punctuation tokens*). Furthermore, we can decide whether to sample *punctuation tokens* by setting the `sample_punctuation` parameter. If `sample_punctuation=False`, then *punctuation tokens* will not be sampled.

Explanation

Let's define the instance we want to explain and verify that the sentiment prediction on the original instance is positive:

```
text = 'This is a good book .'
class_names = ['negative', 'positive']
pred = class_names[predict_fn([text])[0]]
```

Now we can explain the instance:

```
explanation = explainer.explain(text, threshold=0.95)
```

The `explain` method returns an `Explanation` object with the following attributes:

- *anchor*: a list of words in the anchor.
- *precision*: the fraction of times the sampled instances where the anchor holds yields the same prediction as the original instance. The precision will always be \geq `threshold` for a valid anchor.
- *coverage*: the fraction of sampled instances the anchor applies to.

The `raw` attribute is a dictionary which also contains example instances where the anchor holds and the prediction is the same as on the original instance, as well as examples where the anchor holds but the prediction changed to give the user a sense of where the anchor fails. `raw` also stores information on the *anchor*, *precision* and *coverage* of partial

anchors. This allows the user to track the improvement in for instance the *precision* as more features (words in the case of text) are added to the anchor.

Tabular Data

[\[source\]](#)

Initialization and fit

To initialize the explainer, we provide a predict function, a list with the feature names to make the anchors easy to understand as well as an optional mapping from the encoded categorical features to a description of the category. An example for `categorical_names` would be

```
category_map = {0: ["married", "divorced"], 3: ["high school diploma", "master's degree", ""]}
```

Each key in `category_map` refers to the column index in the input for the relevant categorical variable, while the values are lists with the options for each categorical variable. To make it easy, we provide a utility function `gen_category_map` to generate this map automatically from a Pandas dataframe:

```
from alibi.utils import gen_category_map
category_map = gen_category_map(df)
```

Then initialize the explainer:

```
predict_fn = lambda x: clf.predict(preprocessor.transform(x))
explainer = AnchorTabular(predict_fn, feature_names, categorical_names=category_map)
```

The implementation supports one-hot encoding representation of the categorical features by setting `ohe=True`. The `feature_names` and `categorical_names(category_map)` remain unchanged. The prediction function `predict_fn` should expect as input datapoints with one-hot encoded categorical features. To initialize the explainer with the one-hot encoding support:

```
explainer = AnchorTabular(predict_fn, feature_names, categorical_names=category_map,
                           ohe=True)
```

Tabular data requires a fit step to map the ordinal features into quantiles and therefore needs access to a representative set of the training data. `disc_perc` is a list with percentiles used for binning:

```
explainer.fit(X_train, disc_perc=[25, 50, 75])
```

Note that if one-hot encoding support is enabled (`ohe=True`), the `fit` calls expect the data to be one-hot encoded.

Explanation

Let's check the prediction of the model on the original instance and explain:

```
class_names = ['<=50K', '>50K']
pred = class_names[explainer.predict_fn(X)[0]]
explanation = explainer.explain(X, threshold=0.95)
```

The returned `Explanation` object contains the same attributes as the text explainer, so you could explain a prediction as follows:

```
Prediction: <=50K
Anchor: Marital Status = Never-Married AND Relationship = Own-child
Precision: 1.00
Coverage: 0.13
```

Note that if one-hot encoding support is enabled (`ohe=True`), the `explain` calls expect the data to be one-hot encode.

Images

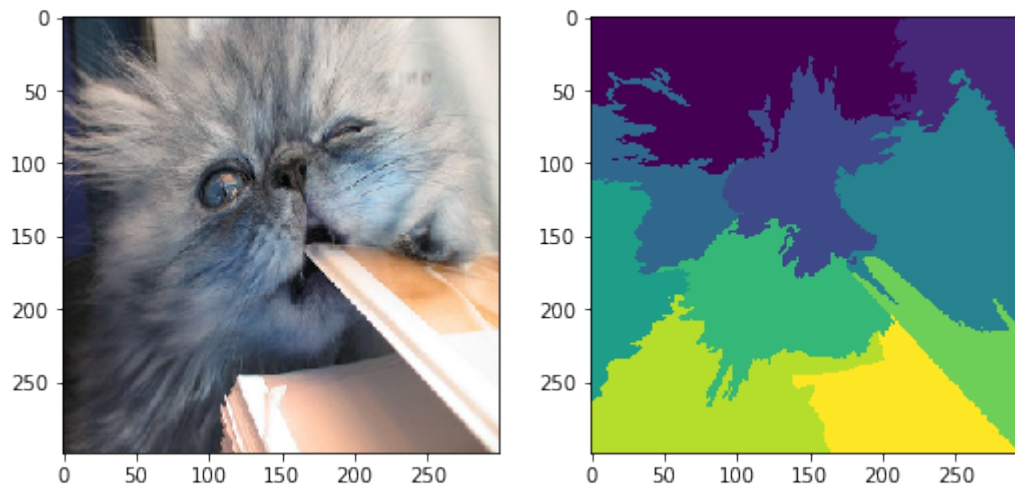
[\[source\]](#)

Initialization

Besides the predict function, we also need to specify either a built in or custom superpixel segmentation function. The built in methods are `felzenszwalb`, `slic` and `quickshift`. It is important to create sensible superpixels in order to speed up convergence and generate interpretable explanations. Tuning the hyperparameters of the segmentation method is recommended.

```
explainer = AnchorImage(predict_fn, image_shape, segmentation_fn='slic',
                        segmentation_kwargs={'n_segments': 15, 'compactness': 20, 'sigma': .5},
                        images_background=None)
```

Example of superpixels generated for the Persian cat picture using the `slic` method:



The following function would be an example of a custom segmentation function dividing the image into rectangles.

```
def superpixel(image, size=(4, 7)):
    segments = np.zeros([image.shape[0], image.shape[1]])
    row_idx, col_idx = np.where(segments == 0)
    for i, j in zip(row_idx, col_idx):
        segments[i, j] = int((image.shape[1]/size[1]) * (i//size[0]) + j//size[1])
    return segments
```

The `images_background` parameter allows the user to provide images used to superimpose on the masked superpixels, not present in the candidate anchor, instead of taking the average value of the masked superpixel. The superimposed

images need to have the same shape as the explained instance.

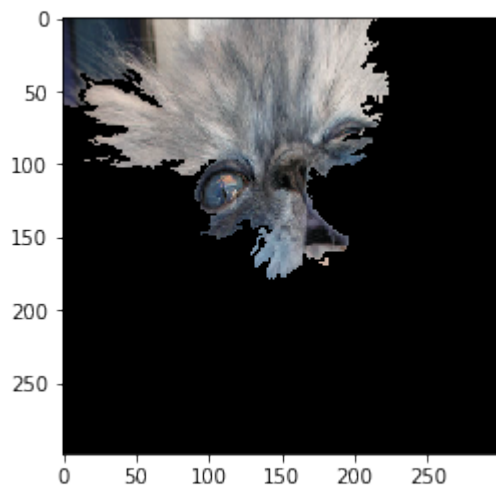
Explanation

We can then explain the instance in the usual way:

```
explanation = explainer.explain(image, p_sample=.5)
```

`p_sample` determines the fraction of superpixels that are either changed to the average superpixel value or that are superimposed.

The `Explanation` object again contains information about the anchor's *precision*, *coverage* and examples where the anchor does or does not hold. On top of that, it also contains a masked image with only the anchor superpixels visible under the *anchor* attribute (see image below) as well as the image's superpixels under *segments*.



7.2.5 Examples

Image

Anchor explanations for ImageNet

Anchor explanations for fashion MNIST

Tabular Data

Anchor explanations on the Iris dataset

Anchor explanations for income prediction

Text

Anchor explanations for movie sentiment

[source]

7.3 Contrastive Explanation Method

Note

To enable support for CEM, you may need to run

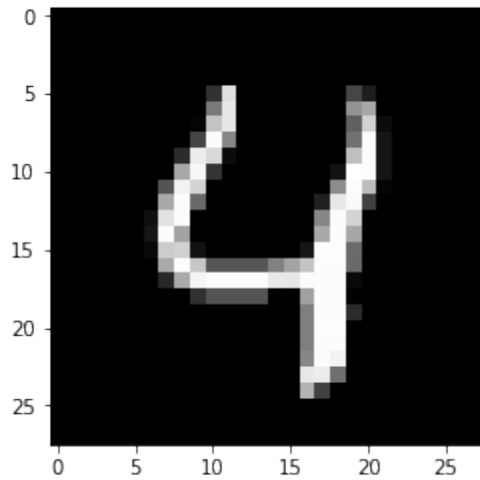
```
pip install alibi[tensorflow]
```

7.3.1 Overview

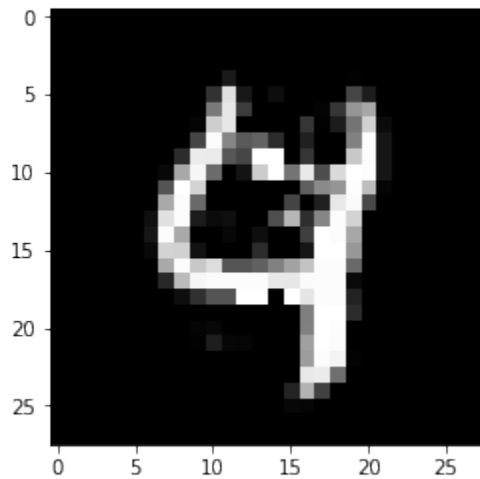
The *Contrastive Explanation Method* (CEM) is based on the paper [Explanations based on the Missing: Towards Contrastive Explanations with Pertinent Negatives](#) and extends the [code](#) open sourced by the authors. CEM generates instance based local black box explanations for classification models in terms of Pertinent Positives (PP) and Pertinent Negatives (PN). For a PP, the method finds the features that should be minimally and sufficiently present (e.g. important pixels in an image) to predict the same class as on the original instance. PN's on the other hand identify what features should be minimally and necessarily absent from the instance to be explained in order to maintain the original prediction class. The aim of PN's is not to provide a full set of characteristics that should be absent in the explained instance, but to provide a minimal set that differentiates it from the closest different class. Intuitively, the Pertinent Positives could be compared to Anchors while Pertinent Negatives are similar to Counterfactuals. As the authors of the paper state, CEM can generate clear explanations of the form: "An input x is classified in class y because features f_i, \dots, f_k are present and because features f_m, \dots, f_p are absent." The current implementation is most suitable for images and tabular data without categorical features.

In order to create interpretable PP's and PN's, feature-wise perturbation needs to be done in a meaningful way. To keep the perturbations sparse and close to the original instance, the objective function contains an elastic net ($\beta L_1 + L_2$) regularizer. Optionally, an auto-encoder can be trained to reconstruct instances of the training set. We can then introduce the L_2 reconstruction error of the perturbed instance as an additional loss term in our objective function. As a result, the perturbed instance lies close to the training data manifold.

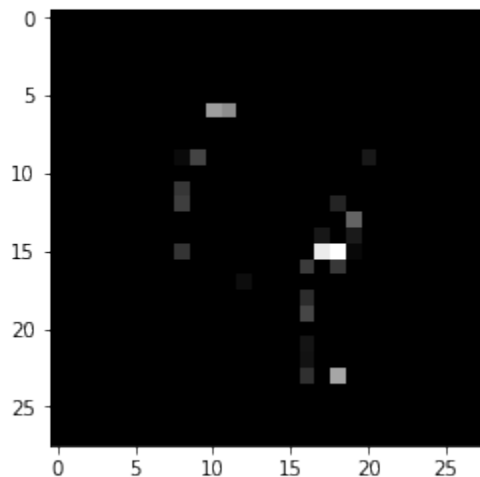
The ability to add or remove features to arrive at respectively PN's or PP's implies that there are feature values that contain no information with regards to the model's predictions. Consider for instance the MNIST image below where the pixels are scaled between 0 and 1. The pixels with values close to 1 define the number in the image while the background pixels have value 0. We assume that perturbations towards the background value 0 are equivalent to removing features, while perturbations towards 1 imply adding features.



It is intuitive to understand that adding features to get a PN means changing 0's into 1's until a different number is formed, in this case changing a 4 into a 9.



To find the PP, we do the opposite and change 1's from the original instance into 0's, the background value, and only keep a vague outline of the original 4.



It is however often not trivial to find these non-informative feature values and domain knowledge becomes very important.

For more details, we refer the reader to the original [paper](#).

7.3.2 Usage

Initialization

The optimizer is defined in TensorFlow (TF) internally. We first load our MNIST classifier and the (optional) auto-encoder. The example below uses Keras or TF models. This allows optimization of the objective function to run entirely with automatic differentiation because the TF graph has access to the underlying model architecture. For models built in different frameworks (e.g. scikit-learn), the gradients of part of the loss function with respect to the input features need to be evaluated numerically. We'll handle this case later.

```
# define models
cnn = load_model('mnist_cnn.h5')
ae = load_model('mnist_ae.h5')
```

We can now initialize the CEM explainer:

```
# initialize CEM explainer
shape = (1,) + x_train.shape[1:]
mode = 'PN'
cem = CEM(cnn, mode, shape, kappa=0., beta=.1,
          feature_range=(x_train.min(), x_train.max()),
          gamma=100, ae_model=ae, max_iterations=1000,
          c_init=1., c_steps=10, learning_rate_init=1e-2,
          clip=(-1000., 1000.), no_info_val=-1.)
```

Besides passing the the predictive and auto-encoder models, we set a number of **hyperparameters** ...

... **general**:

- **mode**: 'PN' or 'PP'.
- **shape**: shape of the instance to be explained, starting with batch dimension. Currently only single explanations are supported, so the batch dimension should be equal to 1.
- **feature_range**: global or feature-wise min and max values for the perturbed instance.

... related to the **optimizer**:

- **max_iterations**: number of loss optimization steps for each value of c ; the multiplier of the first loss term.
- **learning_rate_init**: initial learning rate, follows polynomial decay.
- **clip**: min and max gradient values.

... related to the **non-informative value**:

- **no_info_val**: as explained in the previous section, it is important to define which feature values are considered background and not crucial for the class predictions. For MNIST images scaled between 0 and 1 or -0.5 and 0.5 as in the notebooks, pixel perturbations in the direction of the (low) background pixel value can be seen as removing features, moving towards the non-informative value. As a result, the **no_info_val** parameter is set at a low value like -1. **no_info_val** can be defined globally or feature-wise. For most applications, domain knowledge becomes very important here. If a representative sample of the training set is available, we can always (naively) infer a **no_info_val** by taking the feature-wise median or mean:

```
cem.fit(x_train, no_info_type='median')
```

... related to the **objective function**:

- **c_init** and **c_steps**: the multiplier c of the first loss term is updated for **c_steps** iterations, starting at **c_init**. The first loss term encourages the perturbed instance to be predicted as a different class for a PN and the same class for a PP. If we find a candidate PN or PP for the current value of c , we reduce the value of c for the next optimization cycle to put more emphasis on the regularization terms and improve the solution. If we cannot find a solution, c is increased to put more weight on the prediction class restrictions of the PN and PP before focusing on the regularization.
- **kappa**: the first term in the loss function is defined by a difference between the predicted probabilities for the perturbed instance of the original class and the max of the other classes. $\kappa \geq 0$ defines a cap for this difference, limiting its impact on the overall loss to be optimized. Similar to the original paper, we set κ to 0. in the examples.
- **beta**: β is the L_1 loss term multiplier. A higher value for β means more weight on the sparsity restrictions of the perturbations. Similar to the paper, we set β to 0.1 for the MNIST and Iris datasets.
- **gamma**: multiplier for the optional L_2 reconstruction error. A higher value for γ means more emphasis on the reconstruction error penalty defined by the auto-encoder. Similar to the paper, we set γ to 100 when we have an auto-encoder available.

While the paper's default values for the loss term coefficients worked well for the simple examples provided in the notebooks, it is recommended to test their robustness for your own applications.

Warning

Once a CEM instance is initialized, the parameters of it are frozen even if creating a new instance. This is due to TensorFlow behaviour which holds on to some global state. In order to change parameters of the explainer in the same session (e.g. for explaining different models), you will need to reset the TensorFlow graph manually:

```
import tensorflow as tf
tf.keras.backend.clear_session()
```

You may need to reload your model after this. Then you can create a new CEM instance with new parameters.

Explanation

We can finally explain the instance:

```
explanation = cem.explain(X)
```

The `explain` method returns an `Explanation` object with the following attributes:

- **X**: original instance
- **X_pred**: predicted class of original instance
- **PN** or **PP**: Pertinent Negative or Pertinant Positive
- **PN_pred** or **PP_pred**: predicted class of PN or PP
- **grads_graph**: gradient values computed from the TF graph with respect to the input features at the PN or PP
- **grads_num**: numerical gradient values with respect to the input features at the PN or PP

Numerical Gradients

So far, the whole optimization problem could be defined within the internal TF graph, making autodiff possible. It is however possible that we do not have access to the model architecture and weights, and are only provided with a `predict` function returning probabilities for each class. We initialize the CEM in the same way as before:

```
# define model
lr = load_model('iris_lr.h5')
predict_fn = lambda x: lr.predict(x)

# initialize CEM explainer
shape = (1,) + x_train.shape[1:]
mode = 'PP'
cem = CEM(predict_fn, mode, shape, kappa=0., beta=.1,
          feature_range=(x_train.min(), x_train.max()),
          eps=(1e-2, 1e-2), update_num_grad=100)
```

In this case, we need to evaluate the gradients of the loss function with respect to the input features numerically:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial x}$$

where L is the loss function, p the predict function and x the input features to optimize. There are now 2 additional hyperparameters to consider:

- `eps`: a tuple to define the perturbation size used to compute the numerical gradients. `eps[0]` and `eps[1]` are used respectively for $\delta L / \delta p$ and $\delta p / \delta x$. `eps[0]` and `eps[1]` can be a combination of float values or numpy arrays. For `eps[0]`, the array dimension should be $(1 \times \text{nb of prediction categories})$ and for `eps[1]` it should be $(1 \times \text{nb of features})$. For the Iris dataset, `eps` could look as follows:

```
eps0 = np.array([[1e-2, 1e-2, 1e-2]]) # 3 prediction categories, equivalent to 1e-2
eps1 = np.array([[1e-2, 1e-2, 1e-2, 1e-2]]) # 4 features, also equivalent to 1e-2
eps = (eps0, eps1)
```

- `update_num_grad`: for complex models with a high number of parameters and a high dimensional feature space (e.g. Inception on ImageNet), evaluating numerical gradients can be expensive as they involve prediction calls for each perturbed instance. The `update_num_grad` parameter allows you to set a batch size on which to evaluate the numerical gradients, reducing the number of prediction calls required.

7.3.3 Examples

Contrastive Explanations Method (CEM) applied to MNIST

Contrastive Explanations Method (CEM) applied to Iris dataset

[source]

7.4 Counterfactual Instances

Note

To enable support for counterfactual Instances, you may need to run

```
pip install alibi[tensorflow]
```

7.4.1 Overview

A counterfactual explanation of an outcome or a situation Y takes the form “If X had not occurred, Y would not have occurred” ([Interpretable Machine Learning](#)). In the context of a machine learning classifier X would be an instance of interest and Y would be the label predicted by the model. The task of finding a counterfactual explanation is then to find some X' that is in some way related to the original instance X but leading to a different prediction Y' . Reasoning in counterfactual terms is very natural for humans, e.g. asking what should have been done differently to achieve a different result. As a consequence counterfactual instances for machine learning predictions is a promising method for human-interpretable explanations.

The counterfactual method described here is the most basic way of defining the problem of finding such X' . Our algorithm loosely follows Wachter et al. (2017): [Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR](#). For an extension to the basic method which provides ways of finding higher quality counterfactual instances X' in a quicker time, please refer to [Counterfactuals Guided by Prototypes](#).

We can reason that the most basic requirements for a counterfactual X' are as follows:

- The predicted class of X' is different from the predicted class of X
- The difference between X and X' should be human-interpretable.

While the first condition is straight-forward, the second condition does not immediately lend itself to a condition as we need to first define “interpretability” in a mathematical sense. For this method we restrict ourselves to a particular definition by asserting that X' should be as close as possible to X without violating the first condition. The main issue with this definition of “interpretability” is that the difference between X' and X required to change the model prediction might be so small as to be un-interpretable to the human eye in which case *we need a more sophisticated approach*.

That being said, we can now cast the search for X' as a simple optimization problem with the following loss:

$$L = L_{\text{pred}} + \lambda L_{\text{dist}},$$

where the first loss term L_{pred} guides the search towards points X' which would change the model prediction and the second term λL_{dist} ensures that X' is close to X . This form of loss has a single hyperparameter λ weighing the contributions of the two competing terms.

The specific loss in our implementation is as follows:

$$L(X'|X) = (f_t(X') - p_t)^2 + \lambda L_1(X', X).$$

Here t is the desired target class for X' which can either be specified in advance or left up to the optimization algorithm to find, p_t is the target probability of this class (typically $p_t = 1$), f_t is the model prediction on class t and L_1 is the distance between the proposed counterfactual instance X' and the instance to be explained X . The use of the L_1 distance should ensure that the X' is a sparse counterfactual - minimizing the number of features to be changed in order to change the prediction.

The optimal value of the hyperparameter λ will vary from dataset to dataset and even within a dataset for each instance to be explained and the desired target class. As such it is difficult to set and we learn it as part of the optimization algorithm, i.e. we want to optimize

$$\min_{X'} \max_{\lambda} L(X'|X)$$

subject to

$$|f_t(X') - p_t| \leq \epsilon \text{ (counterfactual constraint),}$$

where ϵ is a tolerance parameter. In practice this is done in two steps, on the first pass we sweep a broad range of λ , e.g. $\lambda \in (10^{-1}, \dots, 10^{-10})$ to find lower and upper bounds $\lambda_{lb}, \lambda_{ub}$ where counterfactuals exist. Then we use bisection to find the maximum $\lambda \in [\lambda_{lb}, \lambda_{ub}]$ such that the counterfactual constraint still holds. The result is a set of counterfactual instances X' with varying distance from the test instance X .

7.4.2 Usage

Initialization

The counterfactual (CF) explainer method works on fully black-box models, meaning they can work with arbitrary functions that take arrays and return arrays. However, if the user has access to a full TensorFlow (TF) or Keras model, this can be passed in as well to take advantage of the automatic differentiation in TF to speed up the search. This section describes the initialization for a TF/Keras model, for fully black-box models refer to [numerical gradients](#).

First we load the TF/Keras model:

```
model = load_model('my_model.h5')
```

Then we can initialize the counterfactual object:

```
shape = (1,) + x_train.shape[1:]
cf = Counterfactual(model, shape, distance_fn='l1', target_proba=1.0,
                    target_class='other', max_iter=1000, early_stop=50, lam_init=1e-1,
                    max_lam_steps=10, tol=0.05, learning_rate_init=0.1,
                    feature_range=(-1e10, 1e10), eps=0.01, init='identity',
                    decay=True, write_dir=None, debug=False)
```

Besides passing the model, we set a number of **hyperparameters** ...

... **general**:

- **shape**: shape of the instance to be explained, starting with batch dimension. Currently only single explanations are supported, so the batch dimension should be equal to 1.
- **feature_range**: global or feature-wise min and max values for the perturbed instance.
- **write_dir**: write directory for Tensorboard logging of the loss terms. It can be helpful when tuning the hyperparameters for your use case. It makes it easy to verify that e.g. not 1 loss term dominates the optimization, that the number of iterations is OK etc. You can access Tensorboard by running `tensorboard --logdir {write_dir}` in the terminal.
- **debug**: flag to enable/disable writing to Tensorboard.

... related to the **optimizer**:

- **max_iterations**: number of loss optimization steps for each value of λ ; the multiplier of the distance loss term.
- **learning_rate_init**: initial learning rate, follows linear decay.

- `decay`: flag to disable learning rate decay if desired
- `early_stop`: early stopping criterion for the search. If no counterfactuals are found for this many steps or if this many counterfactuals are found in a row we change λ accordingly and continue the search.
- `init`: how to initialize the search, currently only "identity" is supported meaning the search starts from the original instance.

... related to the **objective function**:

- `distance_fn`: distance function between the test instance X and the proposed counterfactual X' , currently only "l1" is supported.
- `target_proba`: desired target probability for the returned counterfactual instance. Defaults to 1.0, but it could be useful to reduce it to allow a looser definition of a counterfactual instance.
- `tol`: the tolerance within the `target_proba`, this works in tandem with `target_proba` to specify a range of acceptable predicted probability values for the counterfactual.
- `target_class`: desired target class for the returned counterfactual instance. Can be either an integer denoting the specific class membership or the string `other` which will find a counterfactual instance whose predicted class is anything other than the class of the test instance.
- `lam_init`: initial value of the hyperparameter λ . This is set to a high value $\lambda = 1e^{-1}$ and annealed during the search to find good bounds for λ and for most applications should be fine to leave as default.
- `max_lam_steps`: the number of steps (outer loops) to search for with a different value of λ .

While the default values for the loss term coefficients worked well for the simple examples provided in the notebooks, it is recommended to test their robustness for your own applications.

Warning

Once a `Counterfactual` instance is initialized, the parameters of it are frozen even if creating a new instance. This is due to TensorFlow behaviour which holds on to some global state. In order to change parameters of the explainer in the same session (e.g. for explaining different models), you will need to reset the TensorFlow graph manually:

```
import tensorflow as tf
tf.keras.backend.clear_session()
```

You may need to reload your model after this. Then you can create a new `Counterfactual` instance with new parameters.

Fit

The method is purely unsupervised so no fit method is necessary.

Explanation

We can now explain the instance X :

```
explanation = cf.explain(X)
```

The `explain` method returns an `Explanation` object with the following attributes:

- *cf*: dictionary containing the counterfactual instance found with the smallest distance to the test instance, it has the following keys:
 - *X*: the counterfactual instance
 - *distance*: distance to the original instance
 - *lambda*: value of λ corresponding to the counterfactual
 - *index*: the step in the search procedure when the counterfactual was found
 - *class*: predicted class of the counterfactual
 - *proba*: predicted class probabilities of the counterfactual
 - *loss*: counterfactual loss
- *orig_class*: predicted class of original instance
- *orig_proba*: predicted class probabilities of the original instance
- *all*: dictionary of all instances encountered during the search that satisfy the counterfactual constraint but have higher distance to the original instance than the returned counterfactual. This is organized by levels of λ , i.e. `explanation['all'][0]` will be a list of dictionaries corresponding to instances satisfying the counterfactual condition found in the first iteration over λ during bisection.

Numerical Gradients

So far, the whole optimization problem could be defined within the TF graph, making automatic differentiation possible. It is however possible that we do not have access to the model architecture and weights, and are only provided with a `predict` function returning probabilities for each class. The counterfactual can then be initialized in the same way as before, but using a prediction function:

```
# define model
model = load_model('mnist_cnn.h5')
predict_fn = lambda x: cnn.predict(x)

# initialize explainer
shape = (1,) + x_train.shape[1:]
cf = Counterfactual(predict_fn, shape, distance_fn='l1', target_proba=1.0,
                    target_class='other', max_iter=1000, early_stop=50, lam_init=1e-1,
                    max_lam_steps=10, tol=0.05, learning_rate_init=0.1,
                    feature_range=(-1e10, 1e10), eps=0.01, init
```

In this case, we need to evaluate the gradients of the loss function with respect to the input features X numerically:

$$\frac{\partial L_{\text{pred}}}{\partial X} = \frac{\partial L_{\text{pred}}}{\partial p} \frac{\partial p}{\partial X}$$

where L_{pred} is the predict function loss term, p the predict function and x the input features to optimize. There is now an additional hyperparameter to consider:

- **eps**: a float or an array of floats to define the perturbation size used to compute the numerical gradients of $\delta p / \delta X$. If a single float, the same perturbation size is used for all features, if the array dimension is $(1 \times \text{nb of features})$, then a separate perturbation value can be used for each feature. For the Iris dataset, **eps** could look as follows:

```
eps = np.array([[1e-2, 1e-2, 1e-2, 1e-2]]) # 4 features, also equivalent to eps=1e-2
```

7.4.3 Examples

Counterfactual instances on MNIST

[source]

7.5 Counterfactuals Guided by Prototypes

Note

To enable support for Counterfactuals guided by Prototypes, you may need to run

```
pip install alibi[tensorflow]
```

7.5.1 Overview

This method is based on the [Interpretable Counterfactual Explanations Guided by Prototypes](#) paper which proposes a fast, model agnostic method to find interpretable counterfactual explanations for classifier predictions by using class prototypes.

Humans often think about how they can alter the outcome of a situation. *What do I need to change for the bank to approve my loan?* is a common example. This form of counterfactual reasoning comes natural to us and explains how to arrive at a desired outcome in an interpretable manner. Moreover, examples of counterfactual instances resulting in a different outcome can give powerful insights of what is important to the underlying decision process. This makes it a compelling method to explain predictions of machine learning models. In the context of predictive models, a counterfactual instance describes the necessary change in input features of a test instance that alter the prediction to a predefined output (e.g. a prediction class). The counterfactual is found by iteratively perturbing the input features of the test instance during an optimization process until the desired output is achieved.

A high quality counterfactual instance x_{cf} should have the following desirable properties:

- The model prediction on x_{cf} needs to be close to the predefined output.
- The perturbation δ changing the original instance x_0 into $x_{cf} = x_0 + \delta$ should be sparse.
- The counterfactual x_{cf} should be interpretable. This implies that x_{cf} needs to lie close to both the overall and counterfactual class specific data distribution.
- The counterfactual x_{cf} needs to be found fast enough so it can be used in a real life setting.

We can obtain those properties by incorporating additional loss terms in the objective function that is optimized using gradient descent. A basic loss function for a counterfactual can look like this:

$$Loss = cL_{pred} + \beta L_1 + L_2$$

The first loss term, cL_{pred} , encourages the perturbed instance to predict another class than the original instance. The $\beta L_1 + L_2$ terms act as an elastic net regularizer and introduce sparsity by penalizing the size of the difference between the counterfactual and the perturbed instance. While we can obtain sparse counterfactuals using this objective function, these are often not very interpretable because the training data distribution is not taken into account, and the perturbations are not necessarily meaningful.

The *Contrastive Explanation Method (CEM)* uses an [autoencoder](#) which is trained to reconstruct instances of the training set. We can then add the L_2 reconstruction error of the perturbed instance as a loss term to keep the counterfactual close to the training data distribution. The loss function becomes:

$$Loss = cL_{pred} + \beta L_1 + L_2 + L_{AE}$$

The L_{AE} does however not necessarily lead to interpretable solutions or speed up the counterfactual search. The lack of interpretability occurs because the overall data distribution is followed, but not the class specific one. That's where the prototype loss term L_{proto} comes in. To define the prototype for each prediction class, we can use the encoder part of the previously mentioned autoencoder. We also need the training data or at least a representative sample. We use the model to make predictions on this data set. For each predicted class, we encode the instances belonging to that class. The class prototype is simply the average of the k closest encodings in that class to the encoding of the instance that we want to explain. When we want to generate a counterfactual, we first find the nearest prototype other than the one for the predicted class on the original instance. The L_{proto} loss term tries to minimize the L_2 distance between the counterfactual and the nearest prototype. As a result, the perturbations are guided to the closest prototype, speeding up the counterfactual search and making the perturbations more meaningful as they move towards a typical in-distribution instance. If we do not have a trained encoder available, we can build class representations using [k-d trees](#) for each class. The prototype is then the k nearest instance from a k-d tree other than the tree which represents the predicted class on the original instance. The loss function now looks as follows:

$$Loss = cL_{pred} + \beta L_1 + L_2 + L_{AE} + L_{proto}$$

The method allows us to select specific prototype classes to guide the counterfactual to. For example, in MNIST the closest prototype to a 9 could be a 4. However, we can specify that we want to move towards the 7 prototype and avoid 4.

In order to help interpretability, we can also add a trust score constraint on the proposed counterfactual. The trust score is defined as the ratio of the distance between the encoded counterfactual and the prototype of the class predicted on the original instance, and the distance between the encoded counterfactual and the prototype of the class predicted for the counterfactual instance. Intuitively, a high trust score implies that the counterfactual is far from the originally predicted class compared to the counterfactual class. For more info on trust scores, please check out the [documentation](#).

Because of the L_{proto} term, we can actually remove the prediction loss term and still obtain an interpretable counterfactual. This is especially relevant for fully black box models. When we provide the counterfactual search method with a Keras or TensorFlow model, it is incorporated in the TensorFlow graph and evaluated using automatic differentiation. However, if we only have access to the model's prediction function, the gradient updates are numerical and typically require a large number of prediction calls because of L_{pred} . These prediction calls can slow the search down significantly and become a bottleneck. We can represent the gradient of the loss term as follows:

$$\frac{\partial L_{pred}}{\partial x} = \frac{\partial L_{pred}}{\partial p} \frac{\partial p}{\partial x}$$

where p is the prediction function and x the input features to optimize. For a 28 by 28 MNIST image, the $\delta p / \delta x$ term alone would require a prediction call with batch size $28 \times 28 \times 2 = 1568$. By using the prototypes to guide the search however, we can remove the prediction loss term and only make a single prediction at the end of each gradient update to check whether the predicted class on the proposed counterfactual is different from the original class.

7.5.2 Categorical Variables

It is crucial for many machine learning applications to deal with both continuous numerical and categorical data. Explanation methods which rely on perturbations or sampling of the input features need to make sure those perturbations are meaningful and capture the underlying structure of the data. If not done properly, the perturbed or sampled instances are possibly out of distribution compared to the training data and result in misleading explanations. The perturbation or sampling process becomes tricky for categorical features. For instance random perturbations across possible categories or enforcing a ranking between categories based on frequency of occurrence in the training data do not capture this structure.

Our method first computes the pairwise distances between categories of a categorical variable based on either the model predictions (MVDM) or the context provided by the other variables in the dataset (ABDM). For MVDM, we use the difference between the conditional model prediction probabilities of each category. This method is based on the *Modified Value Difference Metric* (MVDM) by Cost et al (1993). ABDM stands for *Association-Based Distance Metric*, a categorical distance measure introduced by Le et al (2005). ABDM infers context from the presence of other variables in the data and computes a dissimilarity measure based on the Kullback-Leibler divergence. Both methods can also be combined as *ABDM-MVDM*. We can then apply multidimensional scaling to project the pairwise distances into Euclidean space. More details will be provided in a forthcoming paper.

The different use cases are highlighted in the example notebooks linked at the bottom of the page.

7.5.3 Usage

Initialization

The counterfactuals guided by prototypes method works on fully black-box models. This means that they can work with arbitrary functions that take arrays and return arrays. However, if the user has access to a full TensorFlow (TF) or Keras model, this can be passed in as well to take advantage of the automatic differentiation in TF to speed up the search. This section describes the initialization for a TF/Keras model. Please see the [numerical gradients](#) section for black box models.

We first load our MNIST classifier and the (optional) autoencoder and encoder:

```
cnm = load_model('mnist_cnn.h5')
ae = load_model('mnist_ae.h5')
enc = load_model('mnist_enc.h5')
```

We can now initialize the counterfactual:

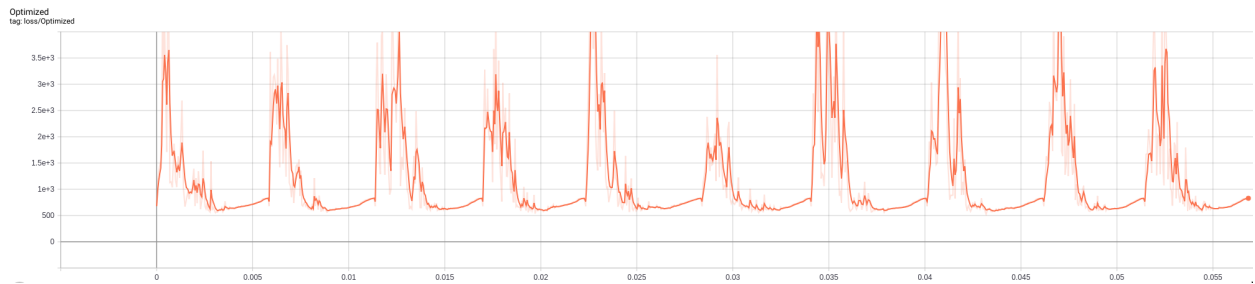
```
shape = (1,) + x_train.shape[1:]
cf = CounterfactualProto(cnn, shape, kappa=0., beta=.1, gamma=100., theta=100.,
                        ae_model=ae, enc_model=enc, max_iterations=500,
                        feature_range=(-.5, .5), c_init=1., c_steps=5,
                        learning_rate_init=1e-2, clip=(-1000., 1000.), write_dir='./cf')
```

Besides passing the predictive, and (optional) autoencoder and models, we set a number of **hyperparameters** ...

... **general**:

- **shape**: shape of the instance to be explained, starting with batch dimension. Currently only single explanations are supported, so the batch dimension should be equal to 1.
- **feature_range**: global or feature-wise min and max values for the perturbed instance.
- **write_dir**: write directory for Tensorboard logging of the loss terms. It can be helpful when tuning the hyperparameters for your use case. It makes it easy to verify that e.g. not 1 loss term dominates the optimization, that the number of iterations is OK etc. You can access Tensorboard by running `tensorboard --logdir`

`{write_dir}` in the terminal. The figure below for example shows the loss to be optimized over different c iterations. It is clear that within each iteration, the number of `max_iterations` steps is too high and we can speed up the search.



... related to the **optimizer**:

- **max_iterations**: number of loss optimization steps for each value of c ; the multiplier of the first loss term.
- **learning_rate_init**: initial learning rate, follows polynomial decay.
- **clip**: min and max gradient values.

... related to the **objective function**:

- **c_init** and **c_steps**: the multiplier c of the first loss term is updated for `c_steps` iterations, starting at `c_init`. The first loss term encourages the perturbed instance to be predicted as a different class than the original instance. If we find a candidate counterfactual for the current value of c , we reduce the value of c for the next optimization cycle to put more emphasis on the other loss terms and improve the solution. If we cannot find a solution, c is increased to put more weight on the prediction class restrictions of the counterfactual.
- **kappa**: the first term in the loss function is defined by a difference between the predicted probabilities for the perturbed instance of the original class and the max of the other classes. $\kappa \geq 0$ defines a cap for this difference, limiting its impact on the overall loss to be optimized. Similar to CEM, we set κ to 0 in the examples.
- **beta**: β is the L_1 loss term multiplier. A higher value for β means more weight on the sparsity restrictions of the perturbations. β equal to 0.1 works well for the example datasets.
- **gamma**: multiplier for the optional L_2 reconstruction error. A higher value for γ means more emphasis on the reconstruction error penalty defined by the autoencoder. A value of 100 is reasonable for the examples.
- **theta**: multiplier for the L_{proto} loss term. A higher θ means more emphasis on the gradients guiding the counterfactual towards the nearest class prototype. A value of 100 worked well for the examples.

When the dataset contains categorical variables, we need to additionally pass the following arguments:

- **cat_vars**: if the categorical variables have ordinal encodings, this is a dictionary with as keys the categorical columns and values the number of categories for the categorical variable in the dataset. If one-hot encoding is applied to the data, then the keys of the `cat_vars` dictionary represent the column where each categorical variable starts while the values still return the number of categories.
- **ohe**: a flag (True or False) whether the categories are one-hot encoded.

It is also important to remember that the perturbations are applied in the numerical feature space, after the categorical variables have been transformed into numerical features. This has to be reflected by the dimension and values of `feature_range`. Imagine for example that we have a dataset with 10 columns. Two of the features are categorical and one-hot encoded. They can both take 3 values each. As a result, the number of columns in the dataset is reduced to 6 when we transform those categorical features to numerical features. As a result, the `feature_range` needs to contain the upper and lower ranges for 6 features.

While the default values for the loss term coefficients worked well for the simple examples provided in the notebooks, it is recommended to test their robustness for your own applications.

Warning

Once a `CounterfactualProto` instance is initialized, the parameters of it are frozen even if creating a new instance. This is due to TensorFlow behaviour which holds on to some global state. In order to change parameters of the explainer in the same session (e.g. for explaining different models), you will need to reset the TensorFlow graph manually:

```
import tensorflow as tf
tf.keras.backend.clear_session()
```

You may need to reload your model after this. Then you can create a new `CounterfactualProto` instance with new parameters.

Fit

If we use an encoder to find the class prototypes, we need an additional `fit` step on the training data:

```
cf.fit(x_train)
```

We also need the `fit` step if the data contains categorical features so we can compute the numerical transformations. In practice, most of these optional arguments do not need to be changed.

```
cf.fit(x_train, d_type='abdm', w=None, disc_perc=[25, 50, 75], standardize_cat_
→vars=False,
    smooth=1., center=True, update_feature_range=True)
```

- `d_type`: the distance metric used to compute the pairwise distances between the categories of each categorical variable. As discussed in the introduction, the options are "abdm", "mvdm" or "abdm-mvdm".
- `w`: if the combined metric "abdm-mvdm" is used, `w` is the weight (between 0 and 1) given to abdm.
- `disc_perc`: for abdm, we infer context from the other features. If there are continuous numerical features present, these are binned according to the quartiles in `disc_perc` before computing the similarity metric.
- `standardize_cat_vars`: whether to return the standardized values for the numerical distances of each categorical feature.
- `smooth`: if the difference in the distances between the categorical variables is too large, then a lower value of the `smooth` argument (0, 1) can smoothen out this difference. This would only be relevant if one categorical variable has significantly larger differences between its categories than others. As a result, the counterfactual search process will likely leave that categorical variable unchanged.
- `center`: whether to center the numerical distances of the categorical variables between the min and max feature ranges.
- `update_feature_range`: whether to update the `feature_range` parameter for the categorical variables based on the min and max values it computed in the `fit` step.

Explanation

We can now explain the instance:

```
explanation = cf.explain(X, Y=None, target_class=None, k=20, k_type='mean',
                       threshold=0., verbose=True, print_every=100, log_every=100)
```

- **X**: original instance
- **Y**: one-hot-encoding of class label for **X**, inferred from the prediction on **X** if *None*.
- **target_class**: classes considered for the nearest class prototype. Either a list with class indices or *None*.
- **k**: number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class.
- **k_type**: use either the average encoding of the **k** nearest instances in a class as the class prototype (**k_type**='mean') or the k-nearest encoding in the class (**k_type**='point'). This parameter is only relevant if an encoder is used to define the prototypes.
- **threshold**: threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements and is rejected.
- **verbose**: if *True*, print progress of counterfactual search every **print_every** steps.
- **log_every**: if **write_dir** for Tensorboard is specified, then log losses every **log_every** steps.

The `explain` method returns an `Explanation` object with the following attributes:

- **cf**: a dictionary with the overall best counterfactual found. `explanation['cf']` has the following *key: value* pairs:
 - **X**: the counterfactual instance
 - **class**: predicted class for the counterfactual
 - **proba**: predicted class probabilities for the counterfactual
 - **grads_graph**: gradient values computed from the TF graph with respect to the input features at the counterfactual
 - **grads_num**: numerical gradient values with respect to the input features at the counterfactual
- **orig_class**: predicted class for original instance
- **orig_proba**: predicted class probabilities for original instance
- **all**: a dictionary with the iterations as keys and for each iteration a list with counterfactuals found in that iteration as values. So for instance, during the first iteration, `explanation['all'][0]`, initially we typically find fairly noisy counterfactuals that improve over the course of the iteration. The counterfactuals for the subsequent iterations then need to be *better* (sparser) than the previous best counterfactual. So over the next few iterations, we probably find less but *better* solutions.

Numerical Gradients

So far, the whole optimization problem could be defined within the TF graph, making automatic differentiation possible. It is however possible that we do not have access to the model architecture and weights, and are only provided with a `predict` function returning probabilities for each class. The counterfactual can then be initialized in the same way:

```
# define model
cnn = load_model('mnist_cnn.h5')
predict_fn = lambda x: cnn.predict(x)
ae = load_model('mnist_ae.h5')
enc = load_model('mnist_enc.h5')

# initialize explainer
shape = (1,) + x_train.shape[1:]
cf = CounterfactualProto(predict_fn, shape, gamma=100., theta=100.,
                        ae_model=ae, enc_model=enc, max_iterations=500,
                        feature_range=(-.5, .5), c_init=1., c_steps=4,
                        eps=(1e-2, 1e-2), update_num_grad=100)
```

In this case, we need to evaluate the gradients of the loss function with respect to the input features numerically:

$$\frac{\partial L_{pred}}{\partial x} = \frac{\partial L_{pred}}{\partial p} \frac{\partial p}{\partial x}$$

where L_{pred} is the loss term related to the prediction function, p is the prediction function and x are the input features to optimize. There are now 2 additional hyperparameters to consider:

- `eps`: a tuple to define the perturbation size used to compute the numerical gradients. `eps[0]` and `eps[1]` are used respectively for $\delta L_{pred} / \delta p$ and $\delta p / \delta x$. `eps[0]` and `eps[1]` can be a combination of float values or numpy arrays. For `eps[0]`, the array dimension should be $(1 \times nb \text{ of prediction categories})$ and for `eps[1]` it should be $(1 \times nb \text{ of features})$. For the Iris dataset, `eps` could look as follows:

```
eps0 = np.array([[1e-2, 1e-2, 1e-2]]) # 3 prediction categories, equivalent to 1e-2
eps1 = np.array([[1e-2, 1e-2, 1e-2, 1e-2]]) # 4 features, also equivalent to 1e-2
eps = (eps0, eps1)
```

- `update_num_grad`: for complex models with a high number of parameters and a high dimensional feature space (e.g. Inception on ImageNet), evaluating numerical gradients can be expensive as they involve prediction calls for each perturbed instance. The `update_num_grad` parameter allows you to set a batch size on which to evaluate the numerical gradients, reducing the number of prediction calls required.

We can also remove the prediction loss term by setting `c_init` to 0 and only run 1 `c_steps`, and still obtain an interpretable counterfactual. This dramatically speeds up the counterfactual search (e.g. by 100x in the MNIST example notebook):

```
cf = CounterfactualProto(predict_fn, shape, gamma=100., theta=100.,
                        ae_model=ae, enc_model=enc, max_iterations=500,
                        feature_range=(-.5, .5), c_init=0., c_steps=1)
```

k-d trees

So far, we assumed that we have a trained encoder available to find the nearest class prototype. This is however not a hard requirement. As mentioned in the *Overview* section, we can use k-d trees to build class representations, find prototypes by querying the trees for each class and return the k nearest class instance as the closest prototype. We can run the counterfactual as follows:

```
cf = CounterfactualProto(cnn, shape, use_kdtree=True, theta=10., feature_range=(-.5, .5))
cf.fit(x_train, trustscore_kwargs=None)
explanation = cf.explain(X, k=2)
```

- `trustscore_kwargs`: keyword arguments for the trust score object used to define the k-d trees for each class. Please check the trust scores [documentation](#) for more info.

7.5.4 Examples

Counterfactuals guided by prototypes on MNIST

Counterfactuals guided by prototypes on California housing dataset

Counterfactual explanations with one-hot encoded categorical variables

Counterfactual explanations with ordinally encoded categorical variables

[source]

7.6 Counterfactuals with Reinforcement Learning

Note

To enable support for Counterfactuals with Reinforcement Learning, you need one of tensorflow or torch installed. You can do so using:

```
pip install alibi[tensorflow]
```

or

```
pip install alibi[torch]
```

7.6.1 Overview

The counterfactual with reinforcement learning is based on the [Model-agnostic and Scalable Counterfactual Explanations via Reinforcement Learning](#) which proposes a fast, model agnostic method to generate batches of counterfactual explanations by replacing the usual optimization procedure with a learnable process. The method **does not** assume model differentiability, relies only on the feedback from the model predictions, allows for target-conditional counterfactual instances and flexible feature range constraints for numerical and categorical features, including the immutability of protected features (e.g, *gender*, *race*). Furthermore, it is easily extendable to multiple data modalities (e.g., images, tabular data).

Counterfactual instances (a.k.a. counterfactual explanations, counterfactuals) are a powerful tool to obtain insight into the underlying decision process exhibited by a black-box model, describing the necessary minimal changes in the input space to alter the prediction towards the desired target. To be of practical use, a counterfactual should be

sparse—close (using some distance measure) to the original instance—and indistinguishable from real instances, that is, it should be in-distribution. Thus, for a loan application system that currently outputs a rejection for a given individual, a counterfactual explanation should suggest plausible minimal changes in the feature values that the applicant could perform to get the loan accepted leading to actionable recourse.

A desirable property of a method for generating counterfactuals is to allow feature conditioning. Real-world datasets usually include immutable features such as *gender* or *race*, which should remain unchanged throughout the counterfactual search procedure. A natural extension of immutability is to restrict a feature to a subset or an interval of values. Thus, following the same loan application example, a customer might be willing to improve their education level from a *High-school graduate* to *Bachelor's* or *Master's*, but not further. Similarly, a numerical feature such as *age* should only increase for a counterfactual to be actionable. To enable such feature conditioning, we propose to use a conditioning vector to guide the generation process.

A counterfactual explanation of a given instance represents a sparse, in-distribution example that alters the model prediction towards a specified target. Let x be the original instance, M a black-box model, $y_M = M(x)$ the model prediction on x and y_T the target prediction. The goal is to produce a counterfactual instance $x_{CF} = x + \delta_{CF}$ where δ_{CF} represents a sparse perturbation vector such that $y_T = M(x_{CF})$. Instead of solving an optimization problem for each input instance, we train a generative model which models the counterfactual instances x_{CF} directly and allows for feature level constraints via an optional conditioning vector c . A conditional counterfactual explanation x_{CF} therefore depends on the tuple $s = (x, y_M, y_T, c)$.

The method **does not** assume the model M to be differentiable and it trains the counterfactual generator using reinforcement learning, namely [Deep Deterministic Policy Gradient \(DDPG\)](#). DDPG interleaves a state-action function approximator called critic (Q), with learning an approximator called actor (μ) to predict the optimal action, which is equivalent to predicting the optimal counterfactual. The method assumes that the critic is differentiable with respect to the action argument, thus allowing to optimize the actor's parameters efficiently through gradient-based methods.

This model-agnostic training pipeline is compatible with various data modalities and only uses sparse model prediction feedback as a reward. For a classification model returning the predicted class label the reward can be defined by an indicator function, $R = 1(M(x_{CF}) = y_T)$. The reward for a regression model, on the other hand is proportional to the proximity of $M(x_{CF})$ to the regression target y_T .

Instead of directly modeling the perturbation vector δ_{CF} in the potentially high-dimensional input space, we first train an [autoencoder](#). The weights of the autoencoder are frozen and μ applies the counterfactual perturbations in the latent space of the encoder. The pre-trained decoder maps the counterfactual embedding back to the input feature space. Since μ operates in the continuous latent space we use the sample efficient DDPG method. We denote by *enc* and *dec* the encoder and the decoder networks, respectively. Given the encoded representation of the input instance $z = enc(x)$, the model prediction y_M , the target prediction y_T and the conditioning vector c , the actor outputs the counterfactual's latent representation $z_{CF} = \mu(z, y_M, y_T, c)$. The decoder then projects the embedding z_{CF} back to the original input space, followed by optional post-processing.

The training step consists of simultaneously optimizing the actor and critic networks. The critic regresses on the reward R determined by the model prediction, while the actor maximizes the critic's output for the given instance through L_{max} . The actor also minimizes two objectives to encourage the generation of sparse, in-distribution counterfactuals. The sparsity loss $L_{sparsity}$ operates on the decoded counterfactual x_{CF} and combines the L_1 loss over the standardized numerical features and the L_0 loss over the categorical ones. The consistency loss $L_{consist}$ aims to encode the counterfactual x_{CF} back to the same latent representation where it was decoded from and helps to produce in-distribution counterfactual instances. Formally, the actor's loss can be written as: $L_{actor} = L_{max} + \lambda_1 L_{sparsity} + \lambda_2 L_{consistency}$

Tabular

	IN	CF	Condition
Age	40	40	[40, 45]
Workclass	Private	Private	{Private, Federal-gov, Self-emp-inc}
Education	High School grad	Masters	{High School grad, Bachelors, Masters}
Marital Status	Married	Married	{Married}
Occupation	Sales	White-Collar	{Sales, White-Collar, Admin}
Relationship	Husband	Husband	{Husband}
Race	White	White	{White}
Sex	Male	Male	{Male}
Capital Gain	0	0	[0, 0]
Capital Loss	0	0	[0, 0]
Hours per week	60	60	[60, 60]
Country	Latin-America	Latin-America	{Latin-America}
Prediction	≤ \$50/y	> \$50k/y	

Samoilescu RF et al., *Model-agnostic and Scalable Counterfactual Explanations via Reinforcement Learning*, 2021

In many real-world applications, some of the input features are immutable, have restricted feature ranges, or are constrained to a subset of all possible feature values. These constraints need to be taken into account when generating actionable counterfactual instances. For instance, *age* and *marital status* could be features in the loan application example. An actionable counterfactual should however only be able to increase the numerical *age* feature and keep the categorical *marital status* feature unchanged. To achieve this we condition the counterfactual generator on a conditioning vector *c*.

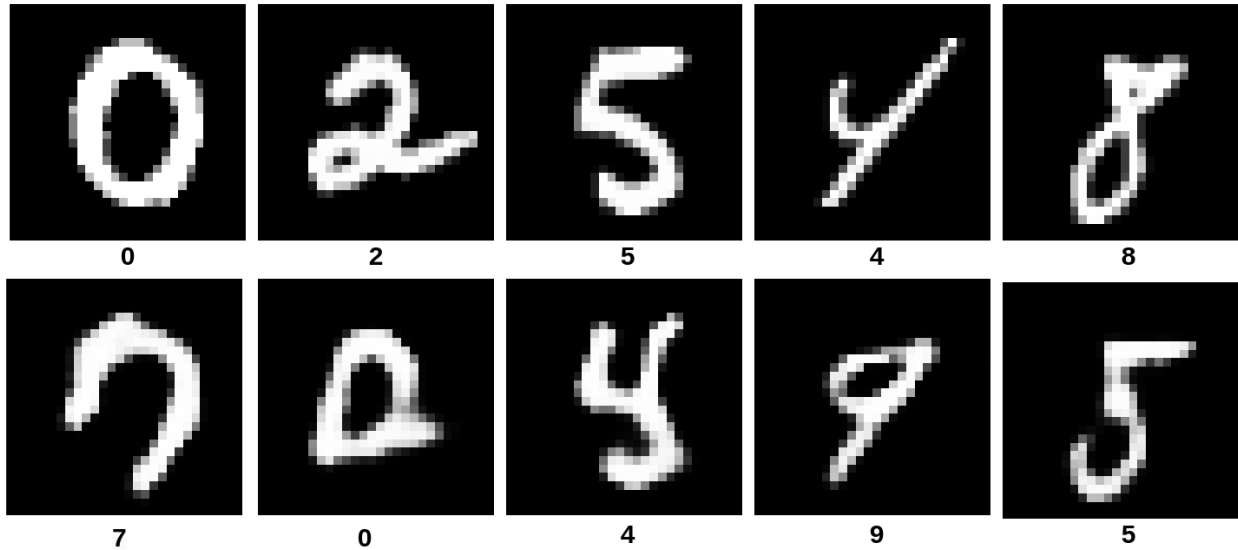
Following the decoding phase, as part of post-processing (denoted by a function *pp*), the numerical values are clipped within the desired range, and categorical values are conditionally sampled according to their masking vector. This step ensures that the generated counterfactual respects the desired feature conditioning before passing it to the model. Note that CFRL is flexible and allows non-differentiable post-processing such as casting features to their original data types (e.g., converting a decoded floating-point *age* to an integer: $40 = \text{int}(40.3)$) and categorical mapping (e.g., *marital status* distribution/one-hot encoding to the married value) since we rely solely on the sparse model prediction reward.

	IN	CF(1)	CF(2)	CF(3)	CF(4)	CF(5)
Age	40	40	40	40	40	40
Workclass	Private	Private	Private	Private	Private	Private
Education	Masters	Dropout	Masters	High School grad	Doctorate	High School grad
Marital Status	Married	Married	Married	Married	Married	Married
Occupation	White-Collar	Service	Professional	Sales	Professional	Military
Relationship	Husband	Husband	Husband	Husband	Husband	Husband
Race	White	White	White	White	White	White
Sex	Male	Male	Male	Male	Male	Male
Capital Gain	0	0	0	0	0	0
Capital Loss	0	0	0	0	0	0
Hours per week	40	40	28	32	28	32
Country	United-States	United-States	United-States	United-States	United-States	United-States
Prediction	> \$50k/y	≤ \$50k/y	≤ \$50k/y	≤ \$50k/y	≤ \$50k/y	≤ \$50k/y

Samoilescu RF et al., *Model-agnostic and Scalable Counterfactual Explanations via Reinforcement Learning*, 2021

Counterfactual diversity is important since it allows the user to take an informed action subject to personal preferences. CFRL can be extended to generate diverse counterfactuals too. Note that the deterministic decoding phase ensures consistency over repeated queries but limits the output to a single possible counterfactual per instance. To increase the diversity, we can sample the conditional vector subject to the user-defined feature constraints. Thus, for unconstrained features, we follow the same sampling procedure applied during training, while for constrained ones, we sample a subset of their restricted values which ensures the feasibility of the counterfactual.

Images



Samoilescu RF et al., *Model-agnostic and Scalable Counterfactual Explanations via Reinforcement Learning*, 2021

CFRL is a flexible method and can be easily extendable to other data modalities, such as images. The training pipeline remains unchanged and only requires a pre-trained autoencoder for each dataset. The method can generate valid, in-distribution counterfactuals even for high-dimensional data.

7.6.2 Usage

CFRL provides a base class specifically designed to be easily adaptable to multiple data modalities. CFRL achieves this flexibility by allowing the user to input custom functions as:

- `reward_func` - element-wise reward function.
- `conditional_func` - generates a conditional vector given a input instance.
- `postprocessing_funcs` - post-processing list of functions. Non-differentiable post-processing can be applied.

For more details, see the documentation [here](#).

Image

We first introduce the image dataset scenario due to ease of usage.

Predictor

Since CFRL works on black-box models, only access to a predict function is needed. The model below is a simple convolutional neural network(CNN) trained on the MNIST dataset:

```
predictor = lambda x: cnn(x)
```

Note that for the classification task the CFRL expects the output of the predictor to be a **2D array**, where the second dimension matches the **number of classes**. The output can be either soft-label distribution (actual probabilities/logits for each class) or hard-label distribution (one-hot encoding). Regardless of the output prediction (logits, probabilities, one-hot encoding), for the classification task the CFRL applies the `argmax` operator on the output.

Autoencoder

CFRL models the perturbation vector δ_{CF} in embedding space, thus a pre-trained autoencoder, `ae`, is required. The autoencoder is a CNN trained on the MNIST dataset, and for simplicity of notation we assume that the model can be factored out in two components, `ae.encoder` and `ae.decoder`, corresponding to the encoder component and decoder component, respectively.

Initialization

```
explainer = CounterfactualRL(predictor=predictor,
                             encoder=ae.encoder,
                             decoder=ae.decoder,
                             coeff_sparsity=COEFF_SPARSITY,
                             coeff_consistency=COEFF_CONSISTENCY,
                             latent_dim=LATENT_DIM,
                             train_steps=300000,
                             batch_size=100,
                             backend="tensorflow")
```

where:

- `predictor` - black-box model.
- `encoder` - pre-trained encoder.
- `decoder` - pre-trained decoder.
- `latent_dim` - embedding/latent dimension.
- `coeff_sparsity` - sparsity loss coefficient.
- `coeff_consistency` - consistency loss coefficient.
- `train_steps` - number of training steps.
- `batch_size` - batch size dimension to be used.
- `backend` - backend to be used. Possible values: `tensorflow`|`pytorch`.

We previously mentioned the CFRL base class can be easily adaptable to multiple data modalities, by allowing the user to specify custom functions. By default, the customizable functions are defined as:

- `reward_func` - by default, checks if the counterfactual prediction label matches the target label.
- `conditional_func` - by default, the function returns `None` which is equivalent to no conditioning.
- `postprocessing_funcs` - by default is an empty list which is equivalent to no post-processing.

Fit

Fitting is straightforward, just passing the training set:

```
explainer.fit(X=X_train)
```

Explanation

We can now explain the instance:

```
explanation = explainer.explain(X=X_test,
                              Y_t=np.array([1]),
                              batch_size=100)
```

where:

- `X` - test instances to be explained.
- `Y_t` - target class. This array can contain either a single entrance that is applied for all test instances or multiple entrances, one for each test instance.
- `batch_size` - batch size to be used at prediction time.

The explain method returns an `Explanation` object with the following attributes:

- `"orig"` - a dictionary containing the following key-value pairs:
 - `"X"` - original input instances.
 - `"class"` - classification labels of the input instances.
- `"cf"` - a dictionary containing the following key-value pairs:
 - `"X"` - counterfactual instances.
 - `"class"` - classification labels of the counterfactual instance.
- `"target"` - target labels.
- `"condition"` - counterfactual conditioning.

Tabular

The tabular scenario follows closely the details provided for the image one, by replacing the custom functions to match the CFRL original implementation.

Predictor

As previously mentioned, CFRL operates in the black-box scenario and thus it can be applied to any model (from differentiable models as neural networks to highly non-differentiable models such as Decision Tree, Random Forest, XGBoost, etc.). The predictor can be defined as:

```
predictor = lambda x: black_box.predict_proba(x)
```

For classification, the output has to respect the same conventions as described in the *Image* section, namely to be a **2D array** having the second dimension match the **number of classes**.

Note that for models that do not support `predict_proba` which outputs the distribution over the classes, one can write a custom function that returns a one-hot encoding representation of the label class without affecting the performance since CFRL applies the `argmax` operator over the output distribution.

Heterogeneous autoencoder

For the heterogeneous autoencoder, `heae`, we use the same naming convention as for *Image* datasets and assume that the autoencoder can be factored into two independent components, `heae.encoder` and `heae.decoder`, representing the encoder and decoder modules. For the tabular scenarios, both the encoder and decoder networks can be fully connected networks.

Since we are dealing with a heterogeneous dataset, that has to be reflected in the output of the decoder. Thus, the convention is that the decoder must be a multiheaded network which implies that the output must be a list of tensors. The first head models all the numerical features (if exist), while the rest of the heads model the categorical ones (one head for each categorical feature).

Heterogeneous datasets require special treatment. In this work, we modeled the numerical features by normal distributions with constant standard deviation and categorical features by categorical distributions. Due to the choice of feature modeling, some numerical features can end up having different types than the original numerical features. For example, a feature like `Age` having the type of `int` can become a `float` due to the autoencoder reconstruction (e.g., `Age=26 -> Age=26.3`). This behavior can be undesirable. Thus we performed a casting when processing the output of the autoencoder (decoder component).

We can specify the datatype of each numerical feature by defining:

```
feature_types = {"Age": int, "Capital Gain": int, "Capital Loss": int, "Hours per week":
    int}
```

(by default each numerical feature is considered to be `float`, thus it can be omitted from the `feature_types` dictionary).

Then we can obtain a heterogeneous autoencoder pre-processor(`heae_preprocessor`) which standardizes the numerical features and transforms the categorical one into a one-hot encoding. The pre-processor is accompanied by an inverse pre-preprocessor(`heae_inv_preprocessor`), designed to map the raw output of the decoder back to the original input space. The inverse pre-processor includes type casting specified in the `feature_types`.

We can obtain the pre-processing pair by:

```
from alibi.explainers.backends.cfml_tabular import get_he_preprocessor
heae_preprocessor, heae_inv_preprocessor = get_he_preprocessor(X=X_train,
    feature_names=adult.feature_names,
    category_map=adult.category_map,
    feature_types=feature_
    types)
```

Constraints

A desirable property of a method for generating counterfactuals is to allow feature conditioning. Real-world datasets usually include immutable features such as *gender* or *race*, which should remain unchanged throughout the counterfactual search procedure. Similarly, a numerical feature such as *age* should only increase for a counterfactual to be actionable.

We define the immutable features as:

```
immutable_features = ['Marital Status', 'Relationship', 'Race', 'Sex']
```

and ranges for numerical attributes as:

```
ranges = {'Age': [0.0, 1.0], 'Hours per week': [-1.0, 0.0], "Capital Gain": [-1.0, 1.0]}
```

The encoding for ranges has the following meaning:

- "Age" - can only increase.
- "Hours per week" - can only decrease.
- "Capital Gain" - can increase or decrease. It is equivalent of saying that there are no constraints, and therefore can be omitted.

Note that the value 0 must be contained in the specified interval. For more details, see the documentation [here](#).

Initialization

```
explainer = CounterfactualRLTabular(predictor=predictor,
                                     encoder=heae.encoder,
                                     decoder=heae.decoder,
                                     latent_dim=LATENT_DIM,
                                     encoder_preprocessor=heae_preprocessor,
                                     decoder_inv_preprocessor=heae_inv_preprocessor,
                                     coeff_sparsity=COEFF_SPARSITY,
                                     coeff_consistency=COEFF_CONSISTENCY,
                                     feature_names=adult.feature_names,
                                     category_map=adult.category_map,
                                     immutable_features=immutable_features,
                                     ranges=ranges,
                                     train_steps=1000000,
                                     batch_size=100,
                                     backend="tensorflow")
```

where:

- decoder - heterogeneous decoder network. The output of the decoder **must be a list of tensors**.
- encoder_preprocessor - heterogeneous autoencoder/encoder pre-processor.
- decoder_inv_preprocessor - heterogeneous autencoder/decoder inverse pre-processor.
- category_map - dataset category mapping. Keys are feature indexes and values are list feature values. Provided by the alibi dataset.
- feature_names - list of feature names. Provided by the alibi dataset.
- ranges - numerical feature ranges, described in the previous section.
- immutable_features - list of immutable features, described in the previous section.

The rest of the arguments were previously described in the *Image* section.

Fit

Similar to the *Image* section, fitting is straight-forward, just passing the training set:

```
explainer.fit(X=X_train)
```

Explanation

Before asking for an explanation, we can define some conditioning:

```
C = [{"Age": [0, 20], "Workclass": ["State-gov", "?", "Local-gov"]}]
```

The above condition is equivalent to say that the `Age` is allowed to increase up to 20 years and that the `Workclass` can change to either "State-gov", "?", "Local-gov" or remain the same. Note that the conditioning is expressed as a δ change from the input and the original feature value will be included by default.

We can generate an explanation by calling the `explain` method as follows:

```
explanation = explainer.explain(X=X_test,
                              Y_t=np.array([1]),
                              C=C,
                              batch_size=BATCH_SIZE)
```

where:

- `C` - conditioning. The list can contain either a single entrance that is applied for all test instances or multiple entrances, one for each test instance.

The rest of the arguments were previously described in the *Image* section.

The `explain` method returns an `Explanation` object described as well in the *Image* section.

Diversity

We can generate a diverse counterfactual set for a single instance by calling the `explain` method and by setting the `diversity=True`:

```
explanation = explainer.explain(X=X_test[0:1],
                              Y_t=np.array([1]),
                              C=C,
                              diversity=True,
                              num_samples=NUM_SAMPLES,
                              batch_size=BATCH_SIZE)
```

where:

- `diversity` - diversity flag.
- `num_samples` - number of distinct counterfactual instances to be generated.

The rest of the arguments were previously described in the *Image* section.

Possible corner case

As we previously mention, tabular scenario requires a heterogeneous decoder. That is a decoder which outputs a list of tensors, one for all numerical features, and one tensor for each of the categorical features. For homogeneous dataset (e.g., all numerical features) the output of the decoder **must be a list that contains one tensor**. One possible workaround is to wrap the decoder as follows:

```
class DecoderList(tf.keras.Model):
    def __init__(self, decoder: tf.keras.Model, **kwargs):
        super().__init__(**kwargs)
        self.decoder = decoder

    def call(self, input: Union[tf.Tensor, List[tf.Tensor]], **kwargs):
        return [self.decoder(input, **kwargs)]

decoder = DecoderList(decoder)
```

Logging

Logging is clearly important when dealing with deep learning models. Thus, we provide an interface to write custom callbacks for logging purposes after each training step which we defined [here](#). In the following section we provide links to notebooks that exemplify how to log using **Weights and Biases**.

Having defined the callbacks, we can define a new explainer that will include logging.

```
import wandb

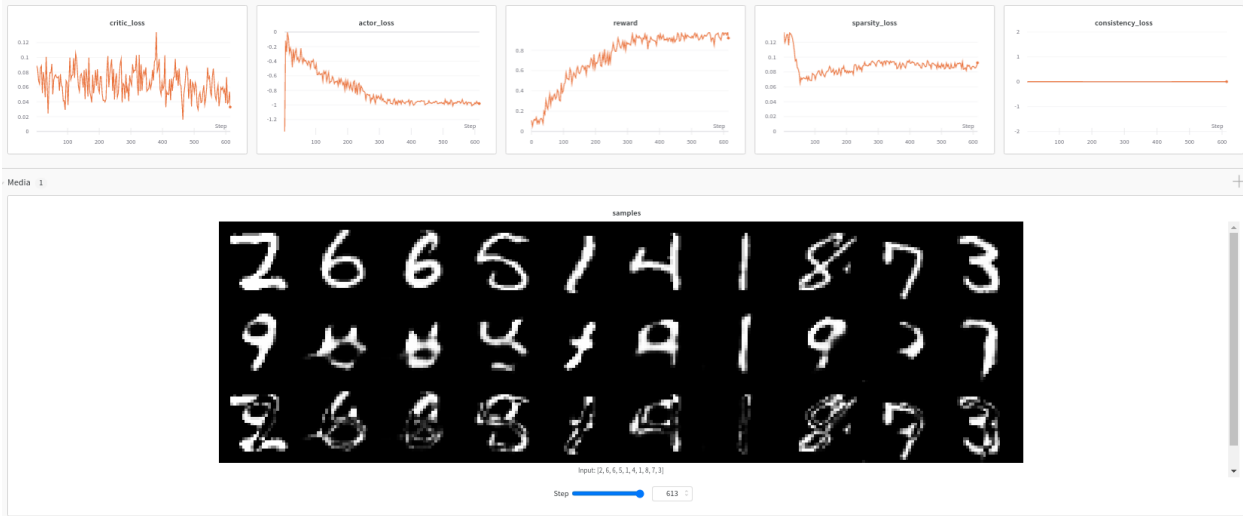
# Initialize wandb.
wandb_project = "Adult Census Counterfactual with Reinforcement Learning"
wandb.init(project=wandb_project)

# Define explainer as before and include callbacks.
explainer = CounterfactualRLTabular(...,
                                     callbacks=[LossCallback(), RewardCallback(),
                                     ↪ TablesCallback()])

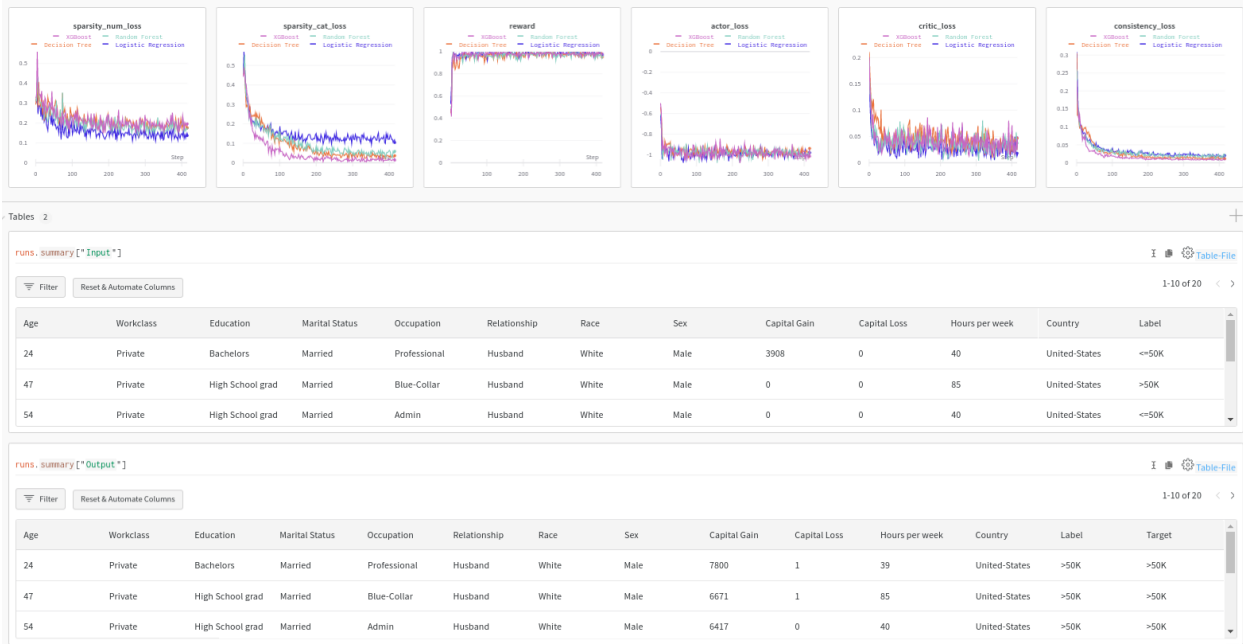
# Fit the explainers.
explainer = explainer.fit(X=X_train)

# Close wandb.
wandb.finish()
```

MNIST Logs



Adult Census Logs



7.6.3 Examples

Counterfactuals with Reinforcement Learning on MNIST

Counterfactuals with Reinforcement Learning on Adult Census

[source]

7.7 Integrated Gradients

Note

To enable support for Integrated Gradients, you may need to run

```
pip install alibi[tensorflow]
```

7.7.1 Overview

Integrated gradients is a method originally proposed in Sundararajan et al., “[Axiomatic Attribution for Deep Networks](#)” that aims to attribute an importance value to each input feature of a machine learning model based on the gradients of the model output with respect to the input. In particular, integrated gradients defines an attribution value for each feature by considering the integral of the gradients taken along a straight path from a baseline instance x' to the input instance x .

7.7.2 Integrated gradients method

The method is applicable to regression and classification models. In the case of a non-scalar output, such as in classification models or multi-target regression, the gradients are calculated for one given element of the output. For classification models, the gradient usually refers to the output corresponding to the true class or to the class predicted by the model.

Let us consider an input instance x , a baseline instance x' and a model $M : X \rightarrow Y$ which acts on the feature space X and produces an output y in the output space Y . Let us now define the function F as

- $F(x) = M(x)$ if the model output is a scalar;
- $F(x) = M_k(x)$ if the model output is a vector, with the index k denoting the k -th element of $M(x)$.

For example, in case of a K -class classification, $M_k(x)$ is the probability of class k , which could be the true class corresponding to x or the highest probability class predicted by the model. The attributions $A_i(x, x')$ for each feature x_i with respect to the corresponding feature x'_i in the baseline are calculated as

$$A_i(x, x') = (x_i - x'_i) \int_0^1 \frac{\partial F(x' + \alpha(x - x'))}{\partial x_i} d\alpha,$$

where the integral is taken along a straight path from the baseline x' to the instance x parameterized by the parameter α .

It is shown that such attributions satisfy the following axioms:

- Sensitivity axiom: if we consider a baseline x' which differs from the input instance x for the value of one feature x_i and yields different predictions, the attribution given to feature x_i must be non-zero.

- Implementation invariance axiom: an attribution method should be such that the attributions do not depend on the particular implementation of the model.
- Completeness axiom: The completeness axiom states that the sum over all features attributions should be equal to the difference between the model output at the instance x and the model output at the baseline x' :

$$\sum_i A_i(x, x') = F(x) - F(x').$$

The proofs that integrated gradients satisfies these axioms are relatively straightforward and are discussed in Sections 2 and 3 of the original paper “[Axiomatic Attribution for Deep Networks](#)”.

7.7.3 Usage

The alibi implementation of the integrated gradients method is specific to TensorFlow and Keras models.

```
import tensorflow as tf
from alibi.explainers import IntegratedGradients

model = tf.keras.models.load_model("path_to_your_model")

ig = IntegratedGradients(model,
                          layer=None,
                          target_fn=None,
                          method="gausslegendre",
                          n_steps=50,
                          internal_batch_size=100)
```

- `model`: Tensorflow or Keras model.
- `layer`: Layer with respect to which the gradients are calculated. If not provided, the gradients are calculated with respect to the input.
- `target_fn`: A scalar function that is applied to the predictions of the model. This can be used to specify which scalar output the attributions should be calculated for (see the example below).
- `method`: Method for the integral approximation. Methods available: `riemann_left`, `riemann_right`, `riemann_middle`, `riemann_trapezoid`, `gausslegendre`.
- `n_steps`: Number of step in the path integral approximation from the baseline to the input instance.
- `internal_batch_size`: Batch size for the internal batching.

```
explanation = ig.explain(X,
                       baselines=None,
                       target=None)

attributions = explanation.attributions
```

- `X`: Instances for which integrated gradients attributions are computed.
- `baselines`: Baselines (starting point of the path integral) for each instance. If the passed value is an `np.ndarray` must have the same shape as `X`. If not provided, all features values for the baselines are set to 0.
- `target`: Defines which element of the model output is considered to compute the gradients. It can be a list of integers or a numeric value. If a numeric value is passed, the gradients are calculated for the same element of the output for all data points. It must be provided if the model output dimension is higher than 1 and no `target_fn`

is provided. For regression models whose output is a scalar, target should not be provided. For classification models target can be either the true classes or the classes predicted by the model.

Example

If your model is a classifier outputting class probabilities (i.e. the predictions are $N \times C$ arrays where N is batch size and C is the number of classes), then you can provide a `target_fn` to the constructor that, for each data point, would select the class of highest probability to calculate the attributions for:

```
from functools import partial
import numpy as np
target_fn = partial(np.argmax, axis=1)
ig = IntegratedGradients(model=model, target_fn=target_fn)
explanation = ig.explain(X)
```

Alternatively, you can leave out `target_fn` and instead provide the predicted class labels directly to the `explain` method:

```
predictions = model.predict(X).argmax(axis=1)
ig = IntegratedGradients(model=model)
explanation = ig.explain(X, target=predictions)
```

Layer attributions

It is possible to calculate the integrated gradients attributions for the model input features or for the elements of an intermediate layer of the model. Specifically,

- If the parameter `layer` is set to its default value `None` as in the example above, the attributions are calculated for each input feature.
- If a layer of the model is passed, the attributions are calculated for each element of the layer passed.

Calculating attribution with respect to an internal layer of the model is particularly useful for models that take text as an input and use word-to-vector embeddings. In this case, the integrated gradients are calculated with respect to the embedding layer (see [example](#) on the IMDB dataset).

Baselines

Conceptually, baselines represent data points which do not contain information useful for the model task, and they are used as a benchmark by the integrated gradients method. Common choices for the baselines are data points with all features values set to zero (for example the black image in case of image classification) or set to a random value.

However, the choice of the baselines can have a significant impact on the values of the attributions. For example, if we consider a simple binary image classification task where a model is trained to predict whether a picture was taken at night or during the day, considering the black image as a baseline would be misleading: in fact, with such a baseline all the dark pixels of the images would have zero attributions, while they are likely to be important for the task at hand.

An extensive discussion about the impact of the baselines on integrated gradients attributions can be found in P. Sturmfels at al., “[Visualizing the Impact of Feature Attribution Baselines](#)”.

Targets

In the context of integrated gradients, the target variable specifies which element of the output should be considered to calculate the attributions. If the output of the model is a scalar, as in the case of single target regression, a target is not necessary, and the gradients are calculated in a straightforward way.

If the output of the model is a vector, the target value specifies the position of the element in the output vector considered for the calculation of the attributions. In case of a classification model, the target can be either the true class or the class predicted by the model for a given input.

7.7.4 Examples

MNIST dataset

Imagenet dataset

IMDB dataset text classification

Text classification using transformers

[source]

7.8 Kernel SHAP

Note

To enable SHAP support, you may need to run:

```
pip install alibi[shap]
```

7.8.1 Overview

The Kernel SHAP (**SH**apley Additive ex**PLAN**ations) algorithm is based on the paper [A Unified Approach to Interpreting Model Predictions](#) by Lundberg et al. and builds on the open source [shap library](#) from the paper's first author.

The algorithm provides model-agnostic (*black box*), human interpretable explanations suitable for regression and classification models applied to tabular data. This method is a member of the *additive feature attribution methods* class; feature attribution refers to the fact that the change of an outcome to be explained (e.g., a class probability in a classification problem) with respect to a *baseline* (e.g., average prediction probability for that class in the training set) can be attributed in different proportions to the model input features.

A simple illustration of the explanation process is shown in Figure 1. Here we see depicted a model which takes as an input features such as Age, BMI or Sex and outputs a continuous value. We know that the average value of that output in a dataset of interest is 0.1. Using the Kernel SHAP algorithm, we attribute the 0.3 difference to the input features. Because the sum of the attribute values equals output - base rate, this method is *additive*. We can see for example that the Sex feature contributes negatively to this prediction whereas the remainder of the features have a positive contribution. For explaining this particular data point, the Age feature seems to be the most important. See our examples on how to perform explanations with this algorithm and visualise the results using the [shap library](#) visualisations [here](#), [here](#) and [here](#).

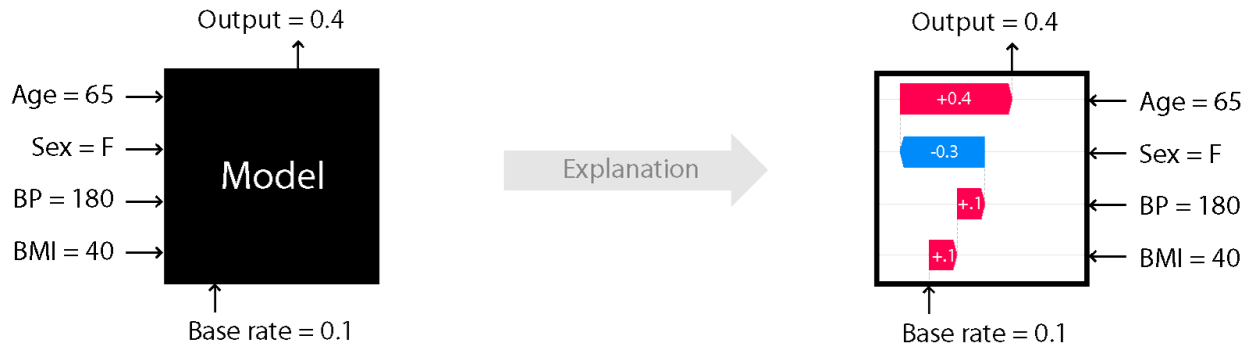


Figure 1: Cartoon illustration of black-box explanation models with Kernel SHAP

Image Credit: Scott Lundberg (see source [here](#))

7.8.2 Usage

In order to compute the shap values , the following hyperparameters can be set when calling the `explain` method:

- `nsamples`: Determines the number of subsets used for the estimation of the shap values. A default of $2 * M + 2 * 11$ is provided where M is the number of features. One is encouraged to experiment with the number of samples in order to determine a value that balances explanation accuracy and runtime.
- `l1_reg`: can take values `0`, `False` to disable, `auto` for automatic regularisation selection, `bic` or `aic` to use ℓ_1 regularised regression with the Bayes/Akaike information criteria for regularisation parameter selection, `num_features(10)` to specify the number of feature effects to be returned or a float value that is used as the regularisation coefficient for the ℓ_1 penalised regression. The default option `auto` uses the least angle regression algorithm with the Akaike Information Criterion if a fraction smaller than `0.2` of the total number of subsets is enumerated.

If the dataset to be explained contains categorical variables, then the following options can be specified *unless* the categorical variables have been grouped (see example below):

- `summarise_result`: if `True`, the shap values estimated for dimensions of an encoded categorical variable are summed and a single shap value is returned for the categorical variable. This requires that both arguments below are specified:
- `cat_var_start_idx`: a list containing the column indices where categorical variables start. For example if the feature matrix contains a categorical feature starting at index `0` and one at index `10`, then `cat_var_start_idx=[0, 10]`.
- `cat_vars_enc_dim`: a list containing the dimension of the encoded categorical variables. The number of columns specified in this list is summed for each categorical variable starting with the corresponding index in `cat_var_start_idx`. So if `cat_var_start_idx=[0, 10]` and `cat_vars_enc_dim=[3, 5]`, then the columns with indices `0, 1 and 2` and `10, 11, 12, 13 and 14` will be combined to return one shap value for each categorical variable, as opposed to 3 and 5.

Explaining continuous datasets

Initialisation and fit

The explainer is initialised by specifying:

- a predict function.
- optionally, setting `link='logit'` if the the model to be explained is a classifier that outputs probabilities. This will apply the logit function to convert outputs to margin space.
- optionally, providing a list of `feature_names`

Hence assuming the classifier takes in 4 inputs and returns probabilities of 3 classes, we initialise its explainer as:

```
from alibi.explainers import KernelShap

predict_fn = lambda x: clf.predict_proba(x)
explainer = KernelShap(predict_fn, link='logit', feature_names=['a', 'b', 'c', 'd'])
```

To fit our classifier, we simply pass our background or ‘reference’ dataset to the explainer:

```
explainer.fit(X_reference)
```

Note that `X_reference` is expected to have a `samples x features` layout.

Explanation

To explain an instance `X`, we simply pass it to the `explain` method:

```
explanation = explainer.explain(X)
```

The returned explanation object has the following fields:

- `explanation.meta`:

```
{'name': 'KernelShap',
 'type': ['blackbox'],
 'explanations': ['local', 'global'],
 'params': {'groups': None,
            'group_names': None,
            'weights': None,
            'summarise_background': False}
}
```

This field contains metadata such as the explainer name and type as well as the type of explanations this method can generate. In this case, the `params` attribute shows that none of the `fit` method optional parameters have been set.

- `explanation.data`:

```
{'shap_values': [array([ 0.8340445 ,  0.12000589, -0.07984099,  0.61758141]),
                 array([-0.71522546,  0.31749045,  0.3146705 , -0.13365639]),
                 array([-0.12984616, -0.47194649, -0.23036243, -0.52314911])],
 'expected_value': array([0.74456904, 1.05058744, 1.15837362]),
 'link': 'logit',
```

(continues on next page)

(continued from previous page)

```

'feature_names': ['a', 'b', 'c', 'd'],
'categorical_names': {},
'raw': {
  'raw_prediction': array([ 2.23635984,  0.83386654, -0.19693058]),
  'prediction': array([0]),
  'instances': array([ 0.93884707, -0.63216607, -0.4350103 , -0.91969562]),
  'importances': {
    '0': {'ranked_effect': array([0.8340445 , 0.61758141, 0.12000589, 0.07984099]),
          'names': ['a', 'd', 'b', 'c']},
    '1': {'ranked_effect': array([0.71522546, 0.31749045, 0.3146705 , 0.13365639]),
          'names': ['a', 'b', 'c', 'd']},
    '2': {'ranked_effect': array([0.52314911, 0.47194649, 0.23036243, 0.12984616]),
          'names': ['d', 'b', 'c', 'a']},
    'aggregated': {'ranked_effect': array([1.67911611, 1.27438691, 0.90944283, 0.
↪ 62487392]),
                   'names': ['a', 'd', 'b', 'c']}
  }
}

```

This field contains:

- **shap_values**: a list of length equal to the number of model outputs, where each entry is an array of dimension `samples x features` of shap values. For the example above , only one instance with 4 features has been explained so the shap values for each class are of dimension `1 x 4`
- **expected_value**: an array of the expected value for each model output across `X_reference`
- **link**: which function has been applied to the model output prior to computing the **expected_value** and estimation of the **shap_values**
- **feature_names**: a list with the feature names, if provided. Defaults to a list containing strings of with the format `feature_{}` if no names are passed
- **categorical_names**: a mapping of the categorical variables (represented by indices in the **shap_values** columns) to the description of the category
- **raw**: this field contains:
 - **raw_prediction**: a `samples x n_outputs` array of predictions for each instance to be explained. Note that this is calculated by applying the link function specified in **link** to the output of **pred_fn**
 - **prediction**: a `samples` array containing the index of the maximum value in the **raw_prediction** array
 - **instances**: a `samples x n_features` array of instances which have been explained
 - **importances**: a dictionary where each entry is a dictionary containing the sorted average magnitude of the shap value (**ranked_effect**) along with a list of feature names corresponding to the re-ordered shap values (**names**). There are `n_outputs + 1` keys, corresponding to `n_outputs` and to the aggregated output (obtained by summing all the arrays in **shap_values**)

Please see our examples on how to visualise these outputs using the shap library visualisations [here](#), [here](#) and [here](#).

Explaining heterogeneous (continuous and categorical) datasets

When the dataset contains both continuous and categorical variables, `categorical_names`, an optional mapping from the encoded categorical features to a description of the category can be passed in addition to the `feature_names` list. This mapping is currently used for determining what type of summarisation should be applied if `X_reference` is large and the fit argument `summarise_background='auto'` or `summarise_background=True` but in the future it might be used for annotating visualisations. The definition of the map depends on what method is used to handle the categorical variables.

By grouping categorical data

By grouping categorical data we estimate a single shap value for each categorical variable.

Initialisation and fit

Assume that we have a dataset with features such as `Marital Status` (first column), `Age` (2nd column), `Income` (3rd column) and `Education` (4th column). The 2nd and 3rd column are continuous variables, whereas the 1st and 4th are categorical ones.

The mapping of categorical variables could be generated from a Pandas dataframe using the utility `gen_category_map`, imported from `alibi.utils`. For this example the output could look like:

```
category_map = {
    0: ["married", "divorced"],
    3: ["high school diploma", "master's degree"],
}
```

Hence, using the same predict function as before, we initialise the explainer as:

```
explainer = KernelShap(
    predict_fn,
    link='logit',
    feature_names=["Marital Status", "Age", "Income", "Education"],
    categorical_names=category_map,
)
```

To group our data, we have to provide the `groups` list, which contains lists with indices that are grouped together. In our case this would be:

```
groups = [[0, 1], [2], [3], [4, 5]]
```

Similarly, the `group_names` are the same as the feature names

```
group_names = ["Marital Status", "Age", "Income", "Education"]
```

Note that, in this case, the keys of the `category_map` are indices into groups. To fit our explainer we pass *one-hot encoded* data to the explainer along with the grouping information.

```
explainer.fit(
    X_reference,
    group_names=group_names,
    groups=groups,
)
```

Explanation

To perform an explanation, we pass *one hot encoded* instances `X` to the `explain` method:

```
explanation = explainer.explain(X)
```

The explanation returned will contain the grouping information in its `meta` attribute

```
{'name': 'KernelShap',
 'type': ['blackbox'],
 'explanations': ['local', 'global'],
 'params': {'groups': [[0, 1], [2], [3], [4, 5]],
            'group_names': ["Marital Status", "Age", "Income", "Education"] ,
            'weights': None,
            'summarise_background': False
          }
}
```

whereas inspecting the `data` attribute shows that one shap value is estimated for each of the four groups:

```
{'shap_values': [array([ 0.8340445 ,  0.12000589, -0.07984099,  0.61758141]),
                 array([-0.71522546,  0.31749045,  0.3146705 , -0.13365639]),
                 array([-0.12984616, -0.47194649, -0.23036243, -0.52314911])],
 'expected_value': array([0.74456904, 1.05058744, 1.15837362]),
 'link': 'logit',
 'feature_names': ["Marital Status", "Age", "Income", "Education"],
 'categorical_names': {},
 'raw': {
   'raw_prediction': array([ 2.23635984,  0.83386654, -0.19693058]),
   'prediction': array([0]),
   'instances': array([ 0.93884707, -0.63216607, -0.4350103 , -0.91969562]),
   'importances': {
     '0': {'ranked_effect': array([0.8340445 , 0.61758141, 0.12000589, 0.07984099]),
           'names': ['a', 'd', 'b', 'c']},
     '1': {'ranked_effect': array([0.71522546, 0.31749045, 0.3146705 , 0.13365639]),
           'names': ['a', 'b', 'c', 'd']},
     '2': {'ranked_effect': array([0.52314911, 0.47194649, 0.23036243, 0.12984616]),
           'names': ['d', 'b', 'c', 'a']},
     'aggregated': {'ranked_effect': array([1.67911611, 1.27438691, 0.90944283, 0.
→ 62487392]),
                    'names': ['a', 'd', 'b', 'c']}
   }
 }
```


By summing output

An alternative to grouping, with a higher runtime cost, is to estimate one shap value for each dimension of the one-hot encoded data and sum the shap values of the encoded dimensions to obtain only one shap value per categorical variable.

Initialisation and fit

The initialisation step is as before:

```
explainer = KernelShap(
    predict_fn,
    link='logit',
    feature_names=["Marital Status", "Age", "Income", "Education"],
    categorical_names=category_map,
)
```

However, note that the keys of the `category_map` have to correspond to the locations of the categorical variables after the effects for the encoded dimensions have been summed up (see details below).

The fit step requires *one hot encoded* data and simply takes the reference dataset:

```
explainer.fit(X_reference)
```

Explanation

To obtain a single shap value per categorical result, we have to specify the following arguments to the `explain` method:

- `summarise_result`: indicates that some shap values will be summed
- `cat_vars_start_idx`: the column indices where the first encoded dimension is for each categorical variable
- `cat_vars_enc_dim`: the length of the encoding dimensions for each categorical variable

```
explanation = explainer.explain(
    X,
    summarise_result=True,
    cat_vars_start_idx=[0, 4],
    cat_vars_enc_dim=[2, 2],
)
```

In our case `Marital Status` starts at column 0 and occupies 2 columns, `Age` and `Income` occupy columns 2 and 3 and `Education` occupies columns 4 and 5.

By combining preprocessor and predictor

Finally, an alternative is to combine the preprocessor and the predictor together in the same object, and fit the explainer on data *before preprocessing*.

Initialisation and fit

To do so, we first redefine our predict function as

```
predict_fn = lambda x: clf.predict(preprocessor.transform(x))
```

The explainer can be initialised as:

```
explainer = KernelShap(
    predict_fn,
    link='logit',
    feature_names=["Marital Status", "Age", "Income", "Education"],
    categorical_names=category_map,
)
```

Then, the explainer should be fitted on *unprocessed* data:

```
explainer.fit(X_referennce_unprocessed)
```

Explanation

We can explain *unprocessed records* simply by calling `explain`:

```
explanation = explainer.explain(X_unprocessed)
```

Running batches of explanations in parallel

Increases in the size of the background dataset, the number of samples used to estimate the shap values or simply explaining a large number of instances dramatically increase the cost of running Kernel SHAP.

To explain batches of instances in parallel, first run `pip install alibi[ray]` to install required dependencies and then simply initialise `KernelShap` specifying the number of physical cores available as follows:

```
distrib_kernel_shap = KernelShap(predict_fn, distributed_opts={'n_cpus': 10})
```

To explain, simply call the `explain` as before - no other changes are required.

Warning

Windows support for the ray Python library is [still experimental](#). Using `KernelShap` in parallel is not currently supported on Windows platforms.

Miscellaneous

Runtime considerations

For a given instance, the runtime of the algorithm depends on:

- the size of the reference dataset
- the dimensionality of the data
- the number of samples used to estimate the shap values

Adjusting the size of the reference dataset

The algorithm automatically warns the user if a background dataset size of more than 300 samples is passed. If the runtime of an explanation with the original dataset is too large, then the algorithm can automatically subsample the background dataset during the `fit` step. This can be achieved by specifying the fit step as

```
explainer.fit(
    X_reference,
    summarise_background=True,
    n_background_samples=150,
)
```

or

```
explainer.fit(
    X_reference,
    summarise_background='auto'
)
```

The auto option will select 300 examples, whereas using the boolean argument allows the user to directly control the size of the reference set. If categorical variables or grouping options are specified, the algorithm uses subsampling of the data. Otherwise, a kmeans clustering algorithm is used to select the background dataset and the samples are weighted according to the frequency of occurrence of the cluster they are assigned to, which is reflected in the `expected_value` attribute of the explainer.

As described above, the explanations are performed with respect to the expected (or weighted-average) output over this dataset so the shap values will be affected by the dataset selection. We recommend experimenting with various ways to choose the background dataset before deploying explanations.

The dimensionality of the data and the number of samples used in shap value estimation

The dimensionality of the data has a slight impact on the runtime, since by default the number of samples used for estimation is $2 \times \text{n_features} + 2^{11}$. In our experiments, we found that either grouping the data or fitting the explainer on unprocessed data resulted in run time savings (but did not run rigorous comparison experiments). If grouping/fitting on unprocessed data alone does not give enough runtime savings, the background dataset could be adjusted. Additionally (or alternatively), the number of samples could be reduced as follows:

```
explanation = explainer.explain(X, nsamples=500)
```

We recommend experimenting with this setting to understand the variance in the shap values before deploying such configurations.

Imbalanced datasets

In some situations, the reference datasets might be imbalanced so one might wish to perform an explanation of the model behaviour around x with respect to $\sum_i w_i f(y_i)$ as opposed to $\mathbb{E}_{\mathcal{D}}[f(y)]$. This can be achieved by passing a list or an 1-D numpy array containing a weight for each data point in `X_reference` as the `weights` argument of the `fit` method.

7.8.3 Theoretical overview

Consider a model f that takes as an input M features. Assume that we want to explain the output of the model f when applied to an input x . Since the model output scale does not have an origin (it is an [affine space](#)), one can only explain the difference of the observed model output with respect to a chosen origin point. This point can be taken to be the function output value for an arbitrary record or the average output over a set of records, \mathcal{D} . Assuming the latter case, for the explanation to be accurate, one requires

$$f(x) - \mathbb{E}_{y \sim \mathcal{D}}[f(y)] = \sum_{i=1}^M \phi_i$$

where \mathcal{D} is also known as a *background dataset* and ϕ_i is the portion of the change attributed to the i th feature. This portion is sometimes referred to as feature importance, effect or simply shap value.

One can conceptually imagine the estimation process for the shap value of the i^{th} feature x_i as consisting of the following steps:

- enumerate all subsets S of the set $F = \{1, \dots, M\} \setminus \{i\}$
- for each $S \subseteq F \setminus \{i\}$, compute the contribution of feature i as $C(i|S) = f(S \cup \{i\}) - f(S)$
- compute the shap value according to

$$\phi_i := \frac{1}{M} \sum_{S \subseteq F \setminus \{i\}} \frac{1}{\binom{M-1}{|S|}} C(i|S).$$

The semantics of $f(S)$ in the above is to compute f by treating \bar{S} as missing inputs. Thus, we can imagine the process of computing the SHAP explanation as starting with S that does not contain our feature, adding feature i and then observing the difference in the function value. For a nonlinear function the value obtained will depend on which features are already in S , so we average the contribution over all possible ways to choose a subset of size $|S|$ and over all subset sizes. The issue with this method is that:

- the summation contains 2^M terms, so the algorithm complexity is $O(M2^M)$
- since most models cannot accept an arbitrary pattern of missing inputs at inference time, calculating $f(S)$ would involve model retraining the model an exponential number of times

To overcome this issue, the following approximations are made:

- the missing features are simulated by replacing them with values from the background dataset
- the feature attributions are estimated instead by solving

$$\min_{\phi_i, \dots, \phi_M} \left\{ \sum_{S \subseteq F} \left[f(S) - \sum_{j \in S} \phi_j \right]^2 \pi_x(S) \right\}$$

where

$$\pi_x(S) = \frac{M-1}{\binom{M}{|S|} |S| (M - |S|)}$$

is the Shapley kernel (Figure 2).

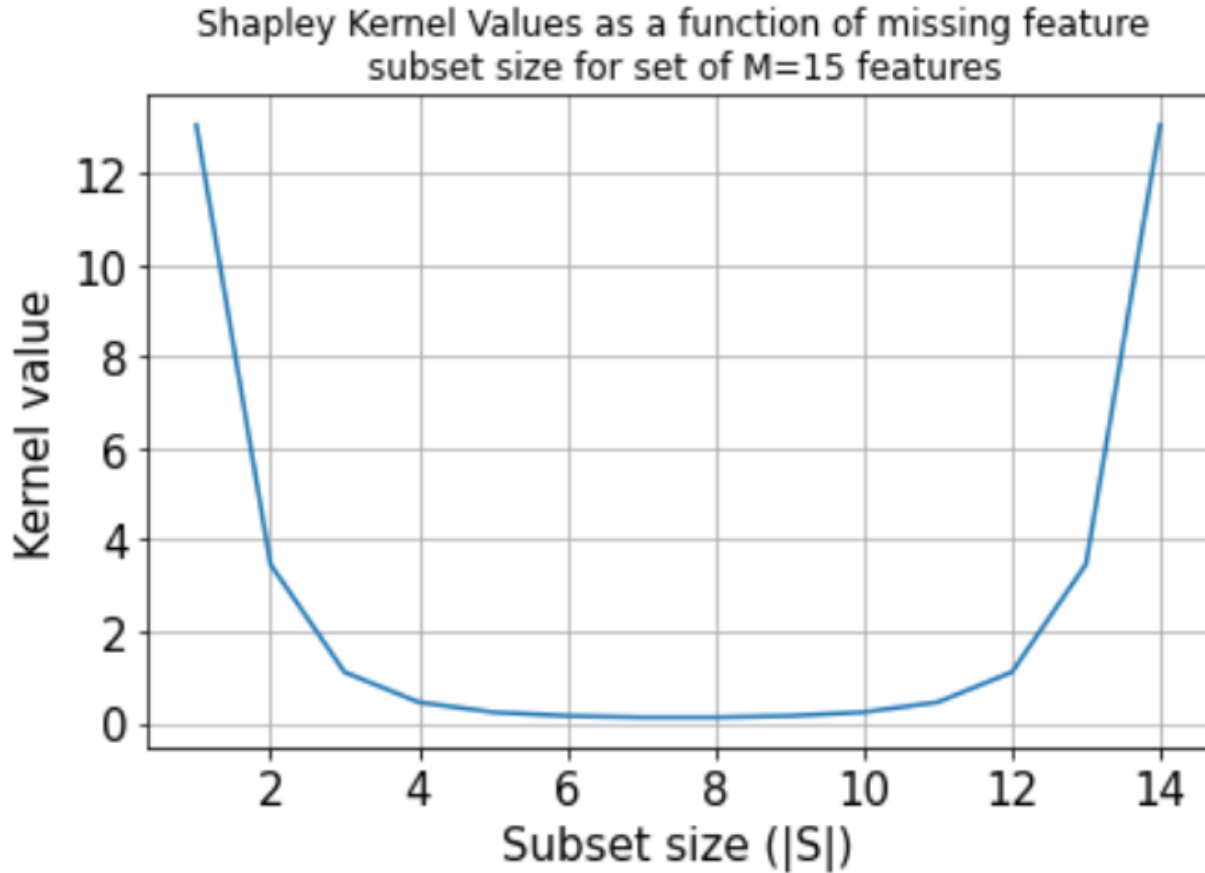


Figure 2: Shapley kernel

Note that the optimisation objective implies above an exponential number of terms. In practice, one considers a finite number of samples n , selecting n subsets S_1, \dots, S_n according to the probability distribution induced by the kernel weights. We can see that the kernel favours either small or large subset sizes, since most of the information about the effect of a particular feature for an outcome change can be obtained by excluding that feature or excluding all the features except for it from the input set.

Therefore, Kernel SHAP returns an approximation of the true Shapley values, whose variability depends on factors such as the size of the structure of the background dataset used to estimate the feature attributions and the number of subsets of missing features sampled. Whenever possible, algorithms specialised for specific model structures (e.g., Tree SHAP, Linear SHAP, integrated gradients) should be used since they are faster and more accurate.

Comparison to other methods

Like [LIME](#), this method provides *local explanations*, in the sense that the attributions are estimated to explain the change from a baseline for a given data point, x . LIME computes the feature attributions by optimising the following objective in order to obtain a locally accurate explanation model (i.e., one that approximates the model to explained well around an instance x):

$$\zeta = \arg \min_{g \in \mathcal{G}} L(f, g, \pi_x) + \Omega(g).$$

Here f is the model to be explained, g is the explanation model (assumed linear), π is a local kernel around instance x (usually cosine or ℓ_2 kernel) and $\Omega(g)$ penalises explanation model complexity. The choices for L , π and Ω in LIME are

heuristic, which can lead to unintuitive behaviour (see [Section 5](#) of Lundberg et al. for a study). Instead, by computing the shap values according to the weighted regression in the previous section, the feature attributions estimated by Kernel SHAP have desirable properties such as *local accuracy*, *consistency* and *missingness*, detailed in [Section 3](#) of Lundberg et al..

Although, in general, local explanations are limited in that it is not clear to what a given explanation applies *around* and instance x (see anchors algorithm overview [here](#) for a discussion), insights into global model behaviour can be drawn by aggregating the results from local explanations (see the work of Lundberg et al. [here](#)). In the future, a distributed version of the Kernel SHAP algorithm will be available in order to reduce the runtime requirements necessary for explaining large datasets.

7.8.4 Examples

Continuous Data

Introductory example: Kernel SHAP on Wine dataset

Comparison with interpretable models

Mixed Data

Handling categorical variables with Kernel SHAP: an income prediction application

Handling categorical variables with Kernel SHAP: fitting explainers on data before pre-processing

Distributed Kernel SHAP: paralelizing explanations on multiple cores

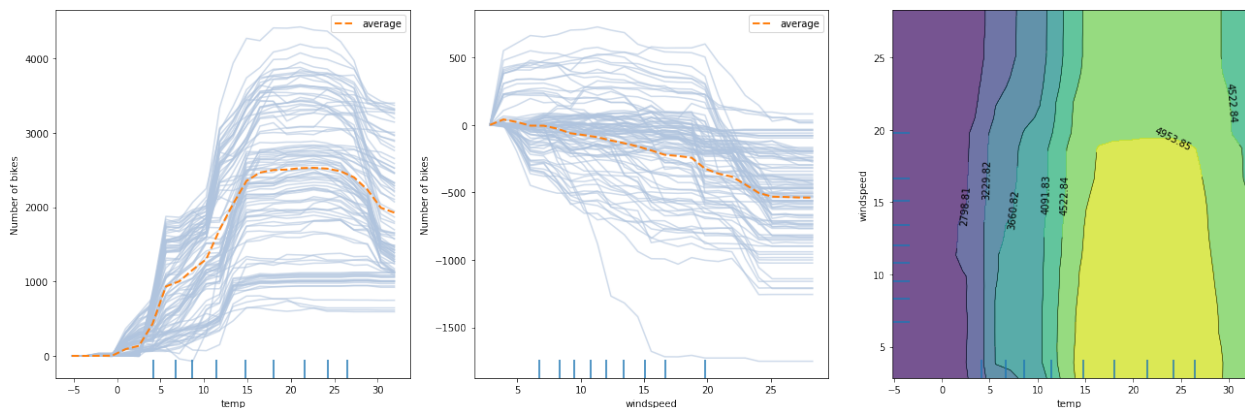
[source]

7.9 Partial Dependence

7.9.1 Overview

The partial dependence (PD) plot proposed by [J.H. Friedman \(2001\)\[1\]](#), is a method to visualize the marginal effect that one or two features have on the predicted outcome of a machine learning model. By inspecting the PD plots, one can understand whether the relation between a feature/pair of features is, for example, a simple linear or quadratic relation, whether it presents a monotonically increasing or decreasing trend, or reveal a more complex response.

The following figure displays two one-way PD plots for two features and a one two-way PD for the same features. The prediction model is random forest regression trained on the [Bike rental\[2\]](#) dataset (see worked [example](#)).



From the left plot, we can observe that the average model prediction increases with the temperature till it reaches approximately 17°C . Then it flattens at a high number until the weather becomes too hot (i.e. approx. 27°C), after which it starts dropping again. A similar analysis can be conducted by inspecting the middle figure for wind speed. As the wind speed increases, fewer and fewer people are riding the bike. Finally, by looking at the right plot, we can visualize how the two features interact. In a few words, the plot suggests that for relatively warm weather, the number of rentals increases as long as the wind is not too rough. For a more detailed analysis, please check the worked [example](#).

For pros & cons of PD plots, see the [Partial Dependence](#) section from the [Introduction](#) materials.

7.9.2 Usage

To initialize the explainer with any black-box model one can directly pass the prediction function and optionally a list of feature names, a list of target names, and a dictionary of categorical names for interpretation and specification of the categorical features:

```
from alibi.explainers import PartialDependence
pd = PartialDependence(predictor=prediction_fn,
                       feature_names=feature_names,
                       categorical_names=categorical_names,
                       target_names=target_names)
```

In addition, similar to the sklearn implementation, alibi supports a faster PD computation for some tree-based sklearn models through the TreePartialDependence explainer. The initialization is similar to the one above, with the difference that the predictor argument will be set to the tree predictor object instead of the prediction function.

```
from alibi.explainer import TreePartialDependence
tree_pd = TreePartialDependence(predictor=tree_predictor,
                                feature_names=feature_names,
                                categorical_names=categorical_names,
                                target_names=target_name)
```

Following the initialization, we can produce an explanation given a dataset X :

```
exp = pd.explain(X=X,
                 features=features,
                 kind='average')
```

Multiple arguments can be provided to the explain method:

- **X** - A $N \times F$ tabular dataset used to calculate partial dependence curves. This is typically the training dataset or a representative sample.
- **features** - An optional list of features or pairs of features for which to calculate the partial dependence. If not provided, the partial dependence will be computed for every single features in the dataset. Some example for features would be: $[0, 2]$, $[0, 2, (0, 2)]$, $[(0, 2)]$, where 0 and 2 correspond to column 0 and 2 in X , respectively.
- **kind** - If set to 'average', then only the partial dependence (PD) averaged across all samples from the dataset is returned. If set to individual, then only the individual conditional expectation (ICE) is returned for each individual from the dataset. Otherwise, if set to 'both', then both the PD and the ICE are returned.
- **percentiles** - Lower and upper percentiles used to limit the feature values to potentially remove outliers from low-density regions. Note that for features with not many data points with large/low values, the PD estimates are less reliable in those extreme regions. The values must be in $[0, 1]$. Only used with **grid_resolution**.
- **grid_resolution** - Number of equidistant points to split the range of each target feature. Only applies if the number of unique values of a target feature in the reference dataset X is greater than the **grid_resolution**

value. For example, consider a case where a feature can take the following values: [0.1, 0.3, 0.35, 0.351, 0.4, 0.41, 0.44, ..., 0.5, 0.54, 0.56, 0.6, 0.65, 0.7, 0.9], and we are not interested in evaluating the marginal effect at every single point as it can become computationally costly (assume hundreds/thousands of points) without providing any additional information for nearby points (e.g., 0.35 and 351). By setting `grid_resolution=5`, the marginal effect is computed for the values [0.1, 0.3, 0.5, 0.7, 0.9] instead, which is less computationally demanding and can provide similar insights regarding the model's behaviour. Note that the extreme values of the grid can be controlled using the `percentiles` argument.

- `grid_points` - Custom grid points. Must be a dict where the keys are the target features indices and the values are monotonically increasing numpy arrays defining the grid points for numerical feature, and a subset of categorical feature values for a categorical feature. If the `grid_points` are not specified, then the grid will be constructed based on the unique target feature values available in the reference dataset `X`, or based on the `grid_resolution` and `percentiles` (check `grid_resolution` to see when it applies). For categorical features, the corresponding value in the `grid_points` can be specified either as numpy array of strings or numpy array of integers corresponding the label encodings. Note that the label encoding must match the ordering of the values provided in the `categorical_names`.

Note that for the tree explainer, we no longer have the option to select the `kind` argument since the `TreePartialDependence` can only compute the PD and not the ICE. In this case, the `explain` call should look like this:

```
tree_exp = tree_pd.explain(X=X,
                          features=features)
```

The rest of the arguments are still available.

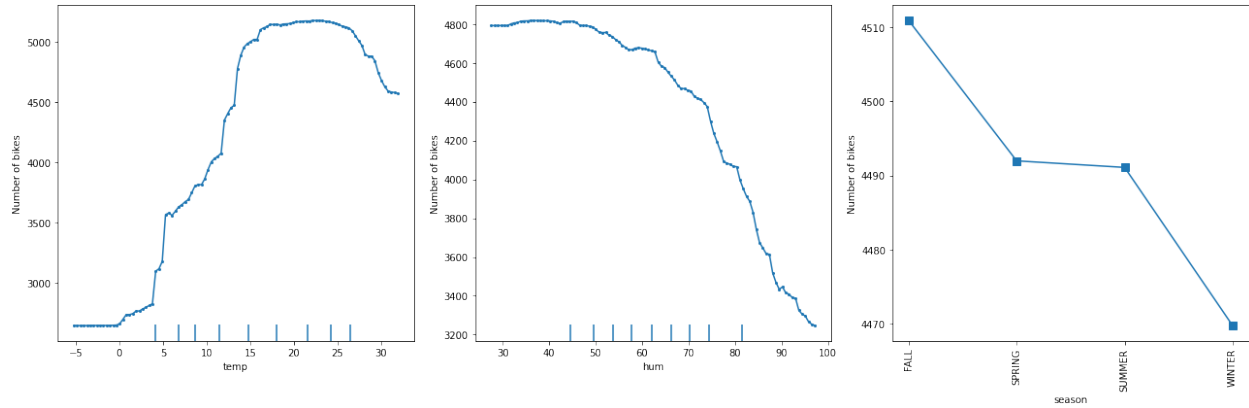
The result `exp/tree_exp` is an `Explanation` object which contains the following data-related attributes:

- `feature_values` - A list of arrays or list of arrays containing the evaluation points for each explained feature passed in the `features` argument (see `explain` method).
- `feature_names` - A list of strings or tuples of string containing the names associated with the explained features elements from `feature_values`.
- `feature_deciles` - a list of arrays (one for each numerical features) of the explained feature deciles.
- `pd_values` - a list of arrays of PD values (one for each feature/pair of features). Each array has a shape of `T x V1 x V2 x ...`, where `T` is the number of target outputs, and `Vi` is the number of evaluation points for the corresponding feature `fi`.
- `ice_values` - a list of arrays of ICE values (one for each feature/pair of feature). Each array has a shape of `T x N x (V1 x V2 x ...)`, where `T` is the number of target outputs, `N` is the number of instances in the reference dataset, and `Vi` is the number of evaluation points for the corresponding feature `fi`. For `TreePartialDependence` the value of this attribute is `None`.

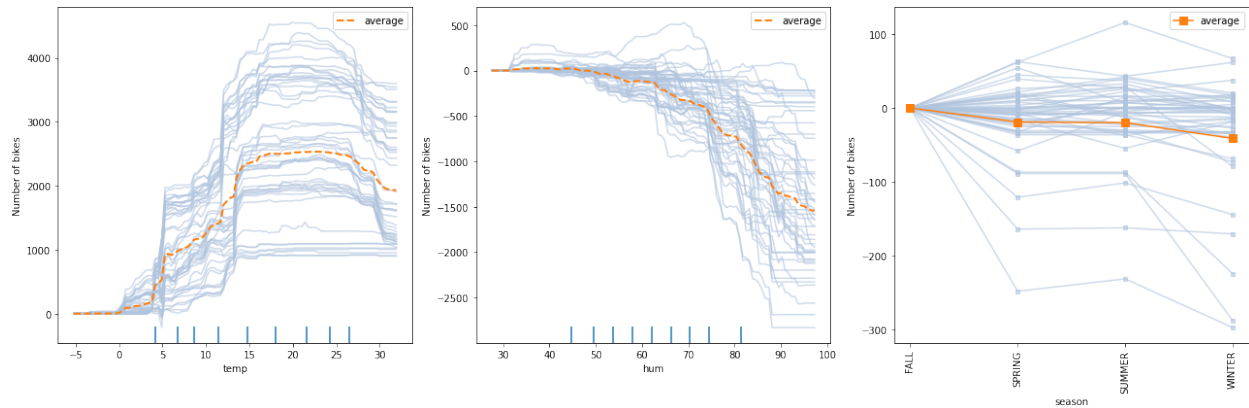
Plotting the `pd_values` and `ice_values` against `exp_feature_values` recovers the PD and the ICE plots, respectively. For convenience we included a plotting function `plot_pd` which automatically produces PD and ICE plots using `matplotlib`.

```
from alibi.explainers import plot_pd
plot_pd(exp)
```

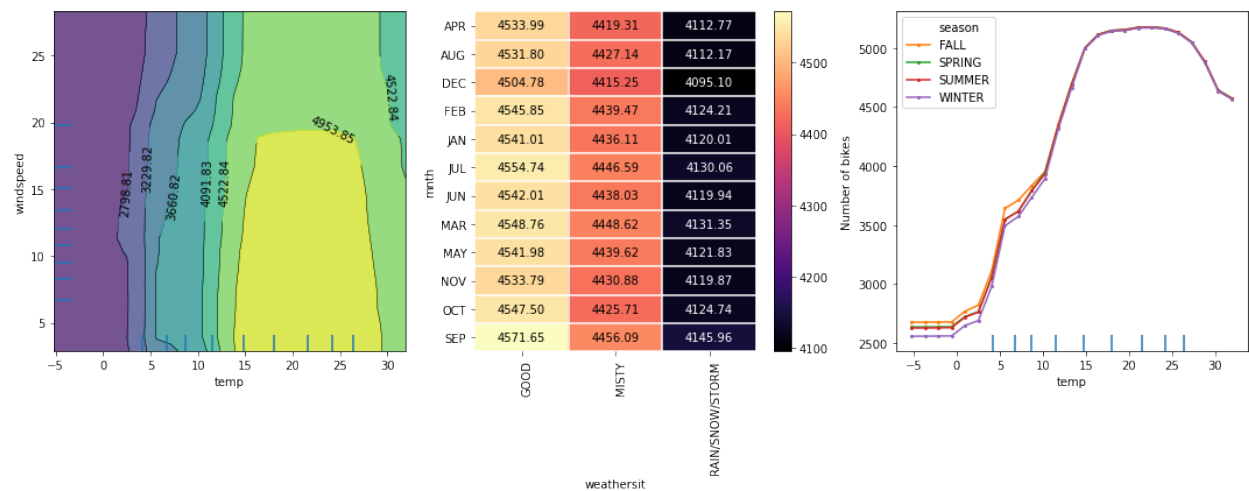
The following figure displays the one way PD plots for a random forest regression trained on the [Bike rental/2/](#) dataset (see worked [example](#)).



The following figure displays the ICE plots for a random forest regression trained on the [Bike rental\[2\]](#) dataset (see worked [example](#)).



The following figure displays the two way PD plots for a random forest regression trained on the [Bike rental\[2\]](#) dataset (see worked [example](#)).



7.9.3 Theoretical exposition

Before diving into the mathematical formulation of the PD, we first introduce some notation. Consider \mathcal{F} to be the set of all features, S be a set of features of interest (i.e. $S \subseteq \mathcal{F}$) that we want to compute the marginal effect for, and C be their complement (i.e. $C = \mathcal{F} \setminus S$). It is important to note that the subset S is not only restricted to one or two features as mentioned in the previous paragraph but can be any subset of the set \mathcal{F} . In practice, though, due to visualization reasons, we will not analyze more than two features at a time.

Given a black-box model, f , we are ready to define the partial dependence for a set of features S as:

$$f_S(x_S) = \mathbb{E}_{X_C}[f(x_S, X_C)] = \int f(x_S, X_C) d\mathbb{P}(X_C),$$

where we denoted random variables by capital letters (e.g., X_C), realizations by lowercase letters (e.g. x_S), and $\mathbb{P}(X_C)$ the probability distribution/measure over the features set C .

In practice, to approximate the integral above, we compute the average over a reference dataset. Formally, let us consider a reference dataset $\mathcal{X} = \{x^{(1)}, x^{(2)}, \dots, x^{(n)}\}$. The PD for the set S can be approximated as:

$$f_S(x_S) = \frac{1}{n} \sum_{i=1}^n f(x_S, x_C^{(i)}).$$

In simple words, to compute the marginal effect of the feature values x_S , we query the model on synthetic instances created from the concatenation of x_S with the feature values $x_C^{(i)}$ from the reference dataset and average the model responses. Already from the computation, we can identify a major limitation of the method. The PD computation assumes feature independence (i.e., features are not correlated) which is a strong and quite restrictive assumption and usually does not hold in practice. A classic example of positively correlated features is *height* and *weight*. If $S = \{\text{height}\}$ and $C = \{\text{weight}\}$, the independence assumption will create an unrealistic synthetic instance, by combining values of *height* and *weight* that are not correlated (e.g. height = 1.85m - height of an adult - and weight = 30kg - weight of a child). This limitation can be addressed by the [Accumulated Local Effects\[3\]](#) (ALE) method.

Although the ALE can handle correlated features, the PD still has some advantages beyond their simple and intuitive definition. The PD can directly be extended to categorical features. For each category of a feature, one can compute the PD by setting all data instance to have the same category and following the same averaging strategy over the reference dataset (i.e., replace features C with feature values from the reference, compute the response, and average all the responses). Note that ALE requires by definition the feature values to be ordinal, which might not be the case for all categorical features. Depending on the context, there exist some methods that allow the ALE to be extended to categorical features for which we recommend the [ALE chapter](#) from the [Interpretable machine learning\[4\]](#) book, as further reading.

We illustrate PD for the simple case of interpreting a linear regression model and demonstrate that it correctly results in showing a linear relationship between the features and the response. To formally prove the linear relationship, consider the following linear regression model:

$$f(x) = \beta_0 + \beta_1 x_1 + \dots \beta_{|\mathcal{F}|} x_{|\mathcal{F}|}.$$

Without loss of generality, we can assume the features of interest come first in the equation above and the rest of the features at the end. We can rewrite the above equation as follows:

$$f(x) = \beta_0 + \beta_1 x_{S_1} + \dots + \beta_{|S|} x_{S_{|S|}} + \beta_{|S|+1} x_{C_1} + \dots + \beta_{|S|+|C|} x_{C_{|C|}},$$

where S_i and C_i represent features in S and C , respectively.

Following the definition of PD, we obtain:

$$f_S(x_S) = \mathbb{E}_{X_C}[f(x_S, X_C)] \quad (7.7)$$

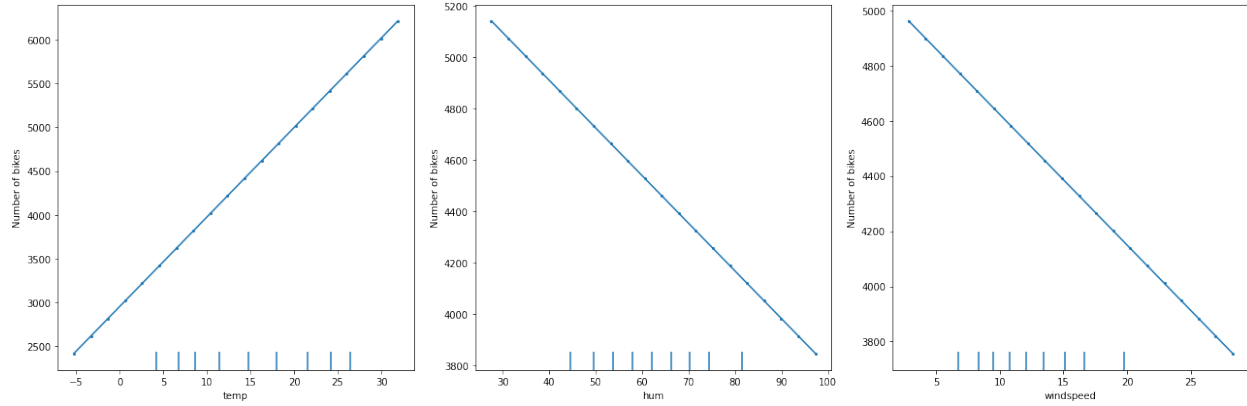
$$= \mathbb{E}_{X_C}[\beta_0 + \beta_1 x_{S_1} + \dots + \beta_{|S|} x_{S_{|S|}} + \beta_{|S|+1} x_{C_1} + \dots + \beta_{|S|+|C|} x_{C_{|C|}}] \quad (7.8)$$

$$= \beta_0 + \beta_1 x_{S_1} + \dots + \beta_{|S|} x_{S_{|S|}} + \mathbb{E}_{X_C}[x_{C_1} + \dots + \beta_{|S|+|C|} x_{C_{|C|}}] \quad (7.9)$$

$$= \beta_0 + \beta_1 x_{S_1} + \dots + \beta_{|S|} x_{S_{|S|}} + K_C \quad (7.10)$$

$$= (\beta_0 + K_C) + \beta_1 x_{S_1} + \dots + \beta_{|S|} x_{S_{|S|}}. \quad (7.11)$$

The following figure displays the PD plots for a linear regression model trained on the [Bike rentals\[2\]](#) dataset:



As expected, we observe a linear relation between the feature value and the marginal effect.

7.9.4 Examples

PD, ICE regression example (Bike rental)

7.9.5 References

[1] Friedman, Jerome H. “Greedy function approximation: a gradient boosting machine.” *Annals of statistics* (2001): 1189-1232.

[2] Fanaee-T, Hadi, and Gama, Joao, ‘Event labeling combining ensemble detectors and background knowledge’, *Progress in Artificial Intelligence* (2013): pp. 1-15, Springer Berlin Heidelberg.

[3] Apley, Daniel W., and Jingyu Zhu. “Visualizing the effects of predictor variables in black box supervised learning models.” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 82.4 (2020): 1059-1086.

[4] Molnar, Christoph. *Interpretable machine learning*. Lulu. com, 2020.

[source]

7.10 Partial Dependence Variance

7.10.1 Overview

Partial Dependence Variance is a method proposed by [Greenwell et al. \(2018\)\[1\]](#) to compute the global feature importance or the feature interaction of a pair of features. As the naming suggests, the feature importance and the feature interactions are summarized in a single positive number given by the variance within the [Partial Dependence \(PD\)\[2, 3\]](#) function. Because the computation relies on the PD, the method is quite intuitive and easy to comprehend.

To get a better intuition of what the proposed method tries to achieve, let us consider a simple example. Given a trained model on the [Bike rental\[4\]](#) dataset, one can compute the PD function for each individual feature. Figure 1 displays the PD for 6 out of 11 features:

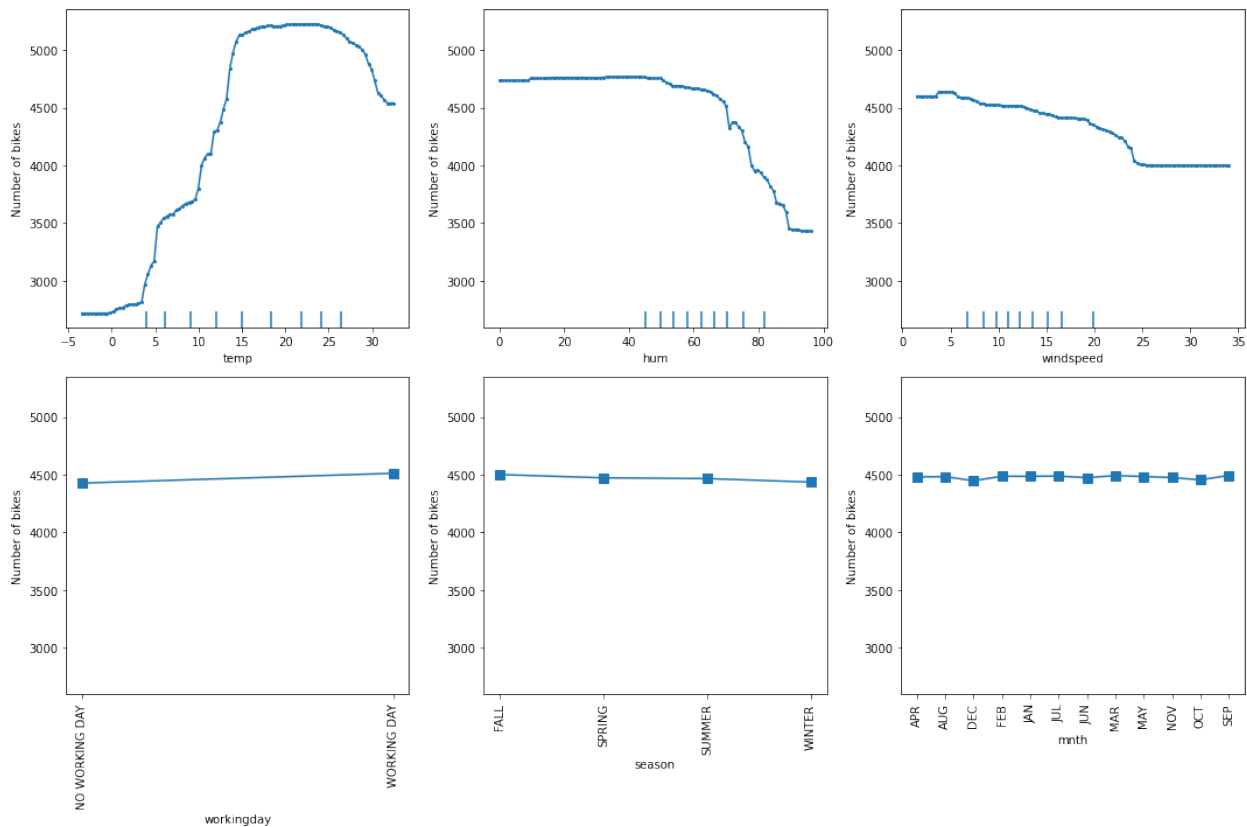


Figure 1. PD plots for Bike rental datasets.*

From the inspection of the plots, we can observe that temperature (temp), humidity (hum), wind speed (windspeed) have a strong non-linear relationship with the predicted outcome. We can observe that the model prediction increases with temperature till it reaches approx $17^{\circ}C$. Then it flattens at a high number until the weather becomes too hot (i.e., approx. $27^{\circ}C$), after which it starts dropping again. The humidity larger than 60% seems to be a factor that inhibits the number of rentals, since we can observe a downward trend from that point onward. Finally, a similar analysis can be conducted for speed. As the wind speed increases, fewer and fewer people are riding the bike.

Quite noticeable are the plots in the second row which show a flat response. Naturally, although some heterogeneity can be hidden, one can assume that the features in the second row have a less impact on the model prediction than the others.

Given the arguments above, one can propose a notion of quantifying the importance of a feature based on a measure of flatness of the PD function, for which the variance represents a natural and straightforward candidate. Figure 2

displays the global feature importance for the given example using the variance of the PD function (left figure) and a model-internal method (i.e., based on impurity because it is a tree ensemble model) (right figure):

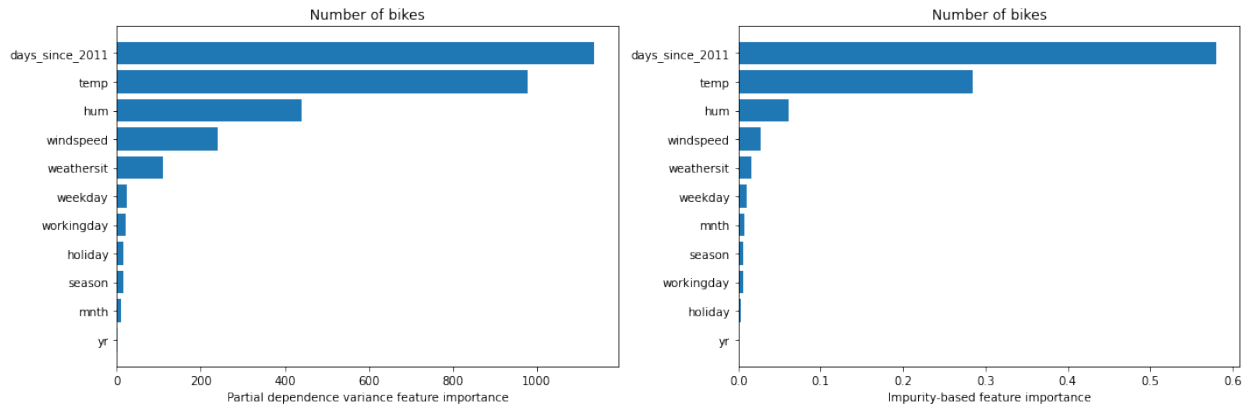


Figure 2. Feature importance comparison between the PD variance (left) and impurity-based method (right).

As we can observe, the two methods agree on the top most salient features.

Advantages:

- the method offers a standardized procedure to quantify the feature importance for any learning algorithm. This contrasts with some internal feature importance notions for some tree-based algorithms such as [Random Forest\[5\]](#) or [Gradient Boosting\[6\]](#), which have their own way to define the importance of a feature.
- the method operates in the black-box regime (i.e., can be applied to any prediction model).
- the method can be adapted to quantify the strength of potential interaction effects.

Drawbacks:

- since the computation of the feature importance is based on the PD, the method captures only the main effect of a feature and ignores possible feature interactions. The PD plot can be flat as the feature affects the predictions mainly through interactions. This is related to the masked heterogeneity.
- the method can fail to detect feature interactions even though those exist (see theoretical overview example below).

7.10.2 Usage

To initialize the explainer with any black-box model, one can directly pass the prediction function and optionally a list of feature names, a list of target names, and a dictionary of categorical names for interpretation and specification of the categorical features.

```
from alibi.explainers import PartialDependenceVariance

pd_variance = PartialDependenceVariance(predictor=predictor_fn,
                                         feature_names=feature_names,
                                         categorical_names=categorical_names,
                                         target_names=target_names)
```

Since the `PartialDependenceVariance` uses the `PartialDependence` explainer, it has support for some tree-based sklearn models directly, just by passing the model to the `predictor` argument (i.e., `predictor=tree_predictor`, where `tree_predictor` is a specific sklearn tree-based model). The rest of the initialization remains the same.

Following the initialization, we can compute two types of explanation for a given dataset X with F features.

The first type of explanation computes feature importance. To compute the feature importance one has to pass the argument `method='importance'` to the `explain` function. The call should look like:

```
exp_importance = pd_variance.explain(X=X, method='importance')
```

By default, the explainer will compute the feature importance for all F features in the dataset. The feature set for which to compute the importance can be customized through the argument `features` as will be presented later.

The second type of explanation computes feature interaction between pairs of features. To compute the feature interaction, one has to pass the argument `method='interaction'` to the `explain` function. The call should look like:

```
exp_interaction = pd_variance.explain(X, method='interaction')
```

By default, the explainer will compute the feature importance for all $F \times (F - 1)$ feature pair combinations from the dataset. As before, the pairs of feature to compute the feature importance for can be customized.

Multiple other arguments can be specified to the `explain` function:

- `X` - A $N \times F$ tabular dataset used to calculate partial dependence curves. This is typically the training dataset or a representative sample.
- `features` - A list of features for which to compute the feature importance or a list of feature pairs for which to compute the feature interaction. Some example of `features` would be: `[0, 1, 3]`, `[(0, 1), (0, 3), (1, 3)]`, where 0, 1, and 3 correspond to the columns 0, 1, and 3 in `X`. If not provided, the feature importance or the feature interaction will be computed for every feature or for every combination of feature pairs, depending on the parameter `method`.
- `method` - Flag to specify whether to compute the feature importance or the feature interaction of the elements provided in `features`. Supported values: `'importance' | 'interaction'`.
- `percentiles` - Lower and upper percentiles used to limit the feature values to potentially remove outliers from low-density regions. Note that for features with not many data points with large/low values, the PD estimates are less reliable in those extreme regions. The values must be in `[0, 1]`. Only used with `grid_resolution`.
- `grid_resolution` - Number of equidistant points to split the range of each target feature. Only applies if the number of unique values of a target feature in the reference dataset `X` is greater than the `grid_resolution` value. For example, consider a case where a feature can take the following values: `[0.1, 0.3, 0.35, 0.351, 0.4, 0.41, 0.44, ..., 0.5, 0.54, 0.56, 0.6, 0.65, 0.7, 0.9]`, and we are not interested in evaluating the marginal effect at every single point as it can become computationally costly (assume hundreds/thousands of points) without providing any additional information for nearby points (e.g., 0.35 and 0.351). By setting `grid_resolution=5`, the marginal effect is computed for the values `[0.1, 0.3, 0.5, 0.7, 0.9]` instead, which is less computationally demanding and can provide similar insights regarding the model's behaviour. Note that the extreme values of the grid can be controlled using the `percentiles` argument.
- `grid_points` - Custom grid points. Must be a dict where the keys are the target features indices and the values are monotonically increasing arrays defining the grid points for a numerical feature, and a subset of categorical feature values for a categorical feature. If the `grid_points` are not specified, then the grid will be constructed based on the unique target feature values available in the dataset `X`, or based on the `grid_resolution` and `percentiles` (check `grid_resolution` to see when it applies). For categorical features, the corresponding value in the `grid_points` can be specified either as array of strings or array of integers corresponding the label encodings. Note that the label encoding must match the ordering of the values provided in the `categorical_names`.

The results `exp` is an `Explanation` object which contains the following data-related attributes:

- `feature_values` - A list of arrays or list of arrays containing the evaluation points for each explained feature passed in the `features` argument (see `explain` method).
- `feature_names` - A list of strings or tuples of string containing the names associated with the explained features elements from `feature_values`.

- `feature_deciles` - a list of arrays (one for each numerical features) of the explained feature deciles.
- `pd_values` - a list of arrays of PD values (one for each feature/pair of features). Each array has a shape of $T \times (V_1 \times [V_2])$, where T is the number of target outputs, and V_i is the number of evaluation points for the corresponding feature f_i .
- `feature_importance` - an array of feature importance for each target and for each explained feature. The array has a size of $T \times F$, where T is the number of targets and F is the number of explained features.
- `feature_interaction` - an array of feature interaction for each target and for each explained feature pair. The array has a size of $T \times FP$, where T is the number of targets and FP is the number of explained feature pairs.
- `conditional_importance_values` - a list of tuples of arrays, where each tuple is associated to a feature pair, and each array inside the tuple corresponds to the conditional feature importance when fixing the value of a feature to a constant and letting the other vary. The arrays inside the tuple have the sizes of $T \times V_1$ and $T \times V_2$, where T is the number of targets, and V_i is the number of evaluation points corresponding to feature f_i .
- `conditional_importance` - a list of tuples of number, where each tuple is associated to a feature pair, and each number inside the tuple corresponds to the conditional feature importance (i.e., taking the importance of the arrays returned in the `conditional_importance_values`). Note that the average of the numbers inside the tuple gives the `feature_interaction` for a feature pair.

Alibi exposes a utility function to plot a summary of the feature importance and feature interaction, or a more detailed exposition of them.

To plot a summary of the feature interaction, one can simply call the `plot_pd_variance` as follows:

```
from alibi.explainers import plot_pd_variance
plot_pd_variance(exp=exp_importance)
```

Figure 3 displays the summary of the feature importance as a horizontal bar plot. By default, the features are sorted in descending order (top to bottom) according to their feature importance.

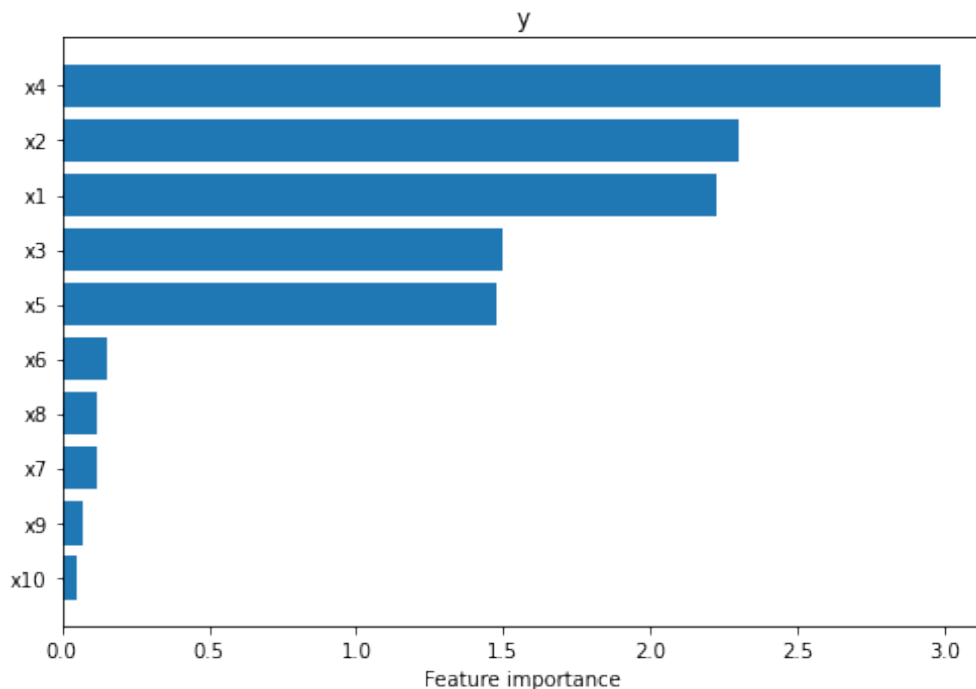


Figure 3. Feature importance summary.

To plot a more detailed exposition of the feature importance, one should set the `summarise=False` flag in the `plot_pd_variance` function. The call should look like:

```
plot_pd_variance(exp=exp_importance, summarise=False)
```

Figure 4 displays the PD plots for the explained features. By default the plots are sorted in descending order (left to right, top to bottom) according to their feature importance:

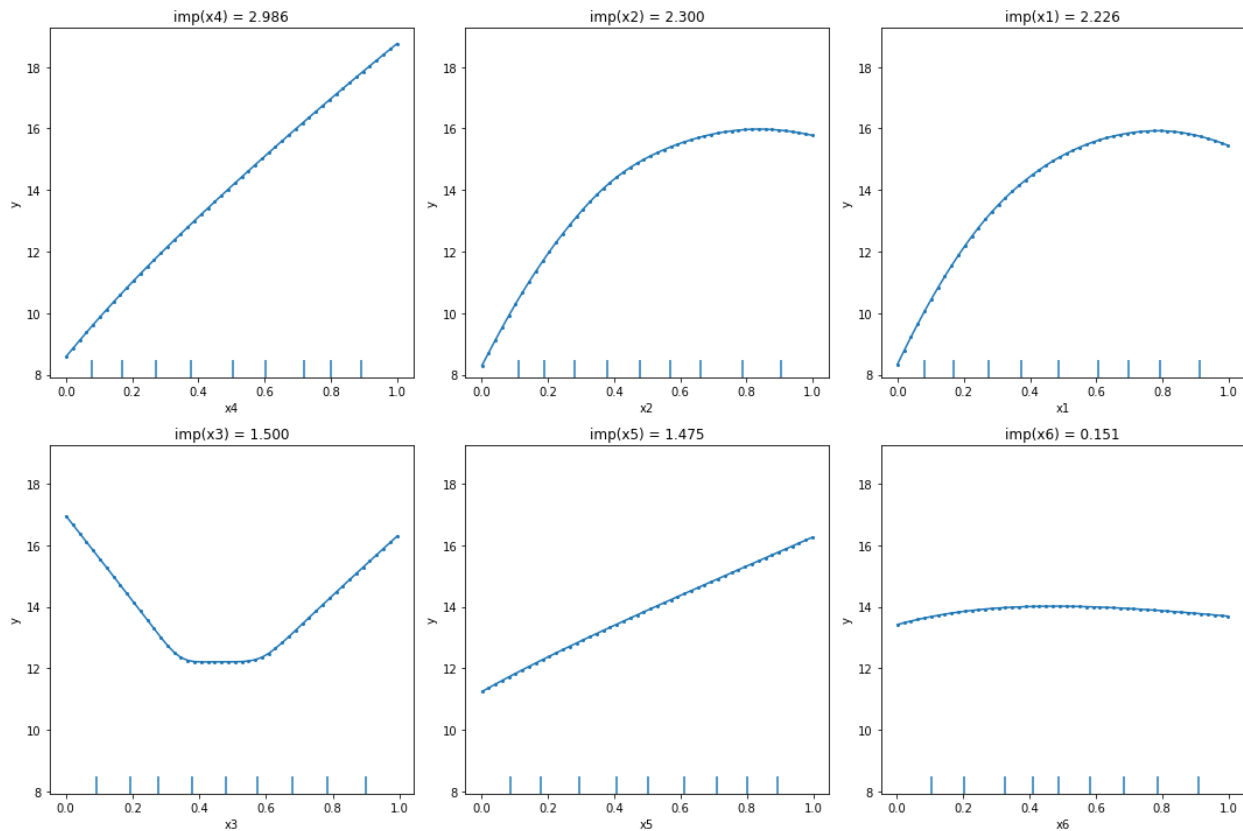


Figure 4. Detailed feature importance plots.

To plot the summary for the feature interaction, we follow the same steps from above:

```
plot_pd_variance(exp=exp_interaction)
```

As before, in Figure 5, the feature interaction is plotted as a horizontal bar plot, sorted in descending order according to the feature interaction.

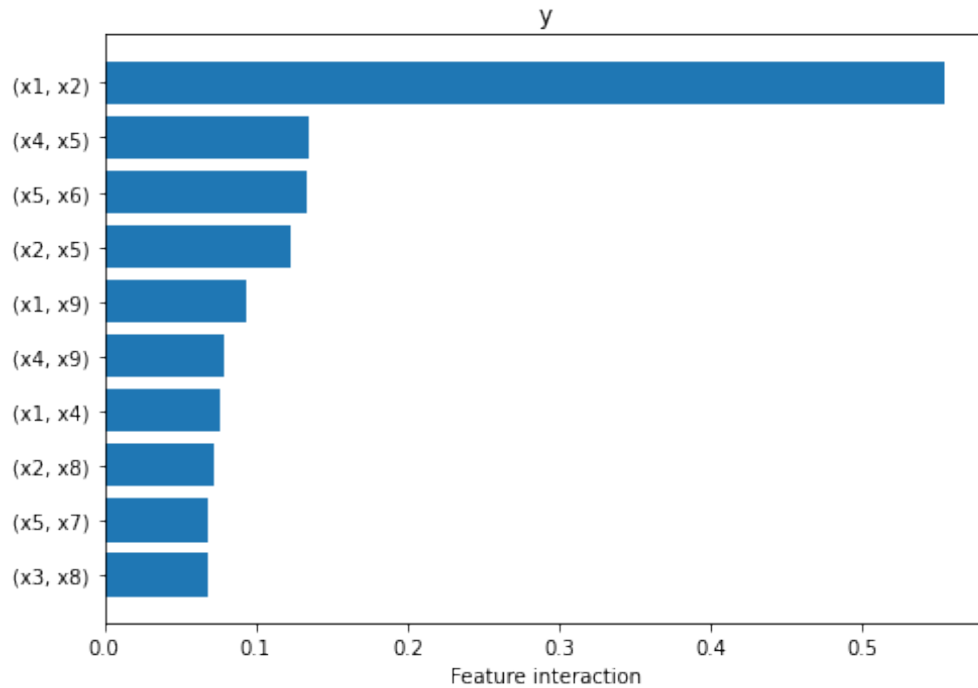


Figure 5. Feature interaction summary.

To plot the more detailed exposition of the feature interaction, we pass, as before, the flag `summarise=False` to the `plot_pd_variance`. It is recommended that the number of axes columns to be divisible by 3 for visualization purposes (see Figure 6).

```
plot_pd_variance(exp=exp_interaction, summarise=False, n_cols=3).
```

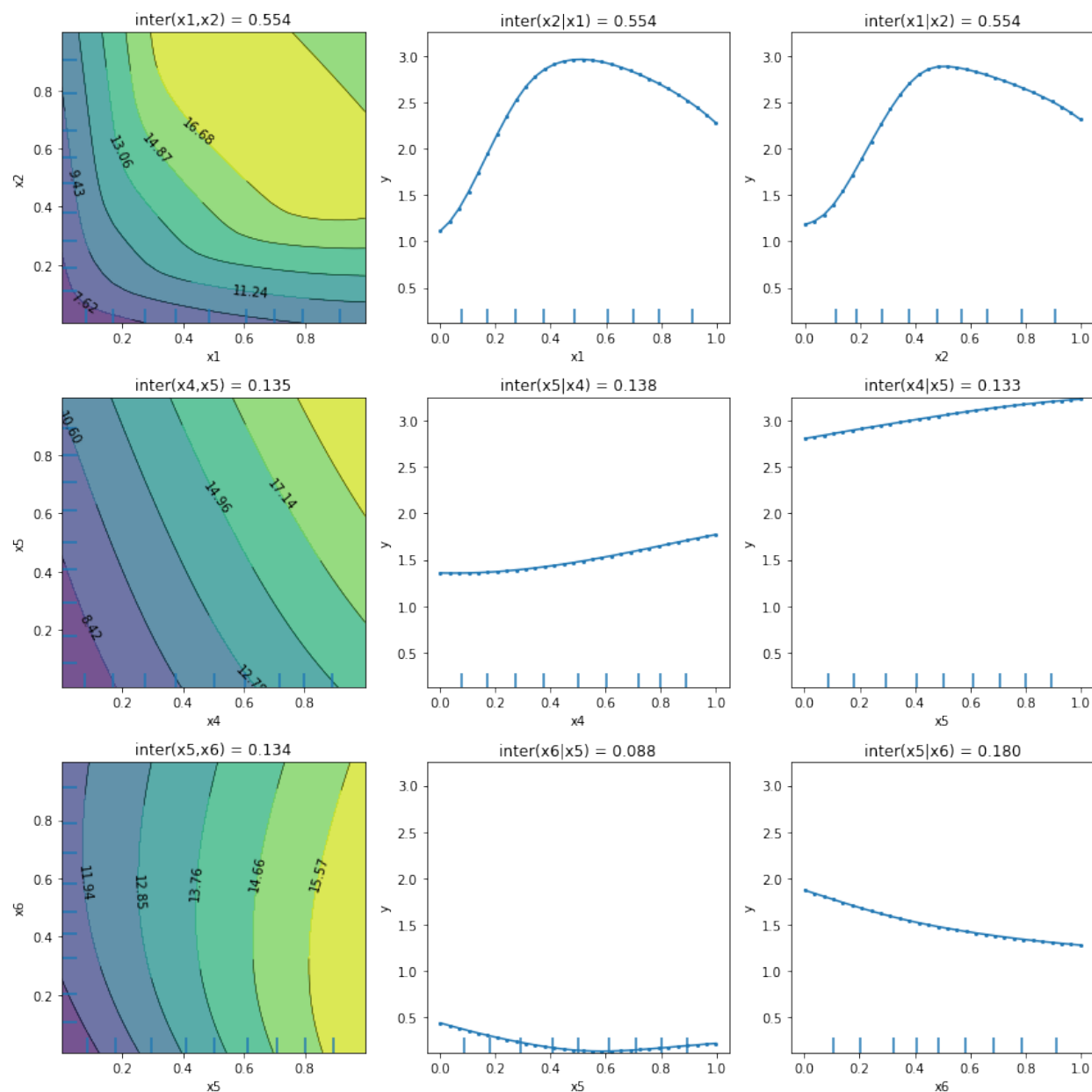


Figure 6. Detailed feature interaction plots.

Note that in this case, for each feature pair, the plots display the 2-way PD function and the two conditional importance plots for each individual feature. By default, the three plots groups are sorted in descending order according to their feature interaction strength.

7.10.3 Theoretical exposition

We split the theoretical exposition in two parts, the first one covering the feature importance and the second one covering the feature interaction.

Feature importance

Following the notation from the [Partial Dependence exposition](#), we say that any variable with a flat PD plot is likely to be less important than those for which the PD plot varies across a wider range. This notion of variable importance is based on a measure of the flatness of the PD function which can be generally stated as:

$$i(x_S) = F(f_S(x_S)),$$

where $F(\cdot)$ is any measure of the “flatness” for the PD of the variables S .

[Greenwell et al. \(2018\)\[1\]](#) proposed to measure the “flatness” as the sample standard deviation for continuous features and the range statistic divided by four for the categorical features. Note that the range divided by four is an estimate of the standard deviation for a small sample size. Formally, those statistics are defined as:

$$i(x_1) = \begin{cases} \sqrt{\frac{1}{k-1} \sum_{i=1}^k [f_1(x_{1i}) - \frac{1}{k} \sum_{j=1}^k f_1(x_{1j})]^2}, & \text{if } x_1 \text{ is continuous} \\ [\max_i(f_1(x_{1i})) - \min_i(f_1(x_{1i}))]/4, & \text{if } x_1 \text{ is categorical} \end{cases} \quad (7.12)$$

Connection to t-statistic

Although the choice of computing the variance of the PD can be motivated intuitively, one can justify it more rigorously by considering a linear model as follows:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p + \epsilon$$

where $\beta_i, i = 1, \dots, p$ are the regression coefficients and $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

To test the significance of a regression coefficient for a least squares problem, one can apply the [t-test](#). For that, one has to compute the [t-statistic](#) given by:

$$\text{t-statistic} = \frac{\hat{\beta}_i - \beta_{H_0}}{s.e.(\hat{\beta}_i)},$$

where $\hat{\beta}_i$ is an estimate of β_i , β_{H_0} is the value under the null hypothesis, and $s.e.$ is the standard error.

If we set $\beta_{H_0} = 0$, then the t-statistic is given by:

$$\text{t-statistic} = \frac{\hat{\beta}_i}{s.e.(\hat{\beta}_i)}.$$

For completeness, we will provide a sketch of the derivation of the t-statistic for the least squares problem. Using the matrix notation, we can rewrite the least squares problem as follows:

$$\hat{\beta} = \arg \min_{\beta \in \mathbb{R}^p} \|Y - X\beta\|_2^2,$$

where Y is the target variable. The least squares solution of the equation above is given by:

$$\hat{\beta} = (X^T X)^{-1} X^T Y.$$

Under the assumption that the true model is given by $Y = X\beta + \epsilon$, we can infer the distribution of $\hat{\beta}$:

$$\hat{\beta} = (X^T X)^{-1} X^T (X\beta + \epsilon) = \beta + (X^T X)^{-1} X^T \epsilon.$$

From the equation above, one can conclude that $\hat{\beta} - \beta \sim \mathcal{N}(0, \sigma^2 (X^T X)^{-1})$. Knowing that $\hat{\beta} - \beta$ has a multivariate normal distribution, we can look at the diagonal entrance and obtain that $\hat{\beta}_i - \beta_i \sim \mathcal{N}(0, \sigma^2 S_{ii})$, where S_{ii} is the i -th diagonal entrance on of the matrix $(X^T X)^{-1}$. The last statement implies that:

$$z_i = \frac{\hat{\beta}_i - \beta_i}{\sqrt{\sigma^2 S_{ii}}} \sim \mathcal{N}(0, 1).$$

Let us denote by $\hat{\epsilon} = (I - X(X^T X)^{-1} X^T)Y$ the residuals, and $s^2 = \frac{\hat{\epsilon}^T \hat{\epsilon}}{n-p}$ be an unbiased estimate of σ^2 . One can show that:

$$V = \frac{(n-p)s^2}{\sigma^2} \sim \chi_{n-p}^2.$$

Given that $z_i \sim \mathcal{N}(0, 1)$ and $V \sim \chi_{n-p}^2$, we can conclude that $t_i = \frac{z_i}{\sqrt{V/(n-p)}}$ is characterized by a t-student distribution with $n-p$ degrees of freedom. With some simple algebraic manipulation, one can show that $t_i = \frac{\hat{\beta}_i - \beta_i}{s.e.(\hat{\beta}_i)}$ as follows:

$$t_i = \frac{\frac{\hat{\beta}_i - \beta_i}{\sqrt{\sigma^2 S_{ii}}}}{\sqrt{\frac{(n-p)s^2}{\sigma^2} / (n-p)}} = \frac{\frac{\hat{\beta}_i - \beta_i}{\sqrt{S_{ii}}}}{\sqrt{s^2}} = \frac{\hat{\beta}_i - \beta_i}{\sqrt{s^2 S_{ii}}} = \frac{\hat{\beta}_i - \beta_i}{s.e.(\hat{\beta}_i)}$$

For a more detailed derivation of the results above, see [this page](#).

To see exactly the connection between the Partial Dependence Variance feature importance and the t-statistic, we consider the following example also presented in [Greenwell et al. \(2018\)/\[1\]](#). Consider the linear model:

$$Y = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2$$

where $\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2$ are constants, X_1 and X_2 are both independent Uniform $[0, 1]$. Since the distributions of X_1 and X_2 are known, one can compute the exact PD $f_i(X_i)$. For example, $f_1(X_1) = \int_0^1 \mathbb{E}[Y|X_1, X_2]p(X_2)dX_2$, where $p(X_2) = 1$. After simple calculus, one obtains:

$$f_1(X_1) = \hat{\beta}_0 + \frac{\hat{\beta}_2}{2} + \hat{\beta}_1 X_1 \text{ and } f_2(X_2) = \hat{\beta}_0 + \frac{\hat{\beta}_1}{2} + \hat{\beta}_2 X_2.$$

Computing the variance for each PD function above, gives us:

$$\mathbb{V}[f_1(X_1)] = \frac{\hat{\beta}_1^2}{12} \text{ and } \mathbb{V}[f_2(X_2)] = \frac{\hat{\beta}_2^2}{12}$$

which implies that the standard deviation is given by $\frac{|\hat{\beta}_1|}{\sqrt{12}}$ and $\frac{|\hat{\beta}_2|}{\sqrt{12}}$, respectively.

On the other hand, the two-tailed t-statistic is given by:

$$t_1 = \frac{|\hat{\beta}_1|}{\sqrt{s^2 S_{11}}} = \frac{|\hat{\beta}_1|}{\sqrt{12s^2}} \text{ and } t_2 = \frac{|\hat{\beta}_2|}{\sqrt{s^2 S_{22}}} = \frac{|\hat{\beta}_2|}{\sqrt{12s^2}}$$

which matches the Partial Dependence Variance up to a proportionality constant. Thus, the variance of the PD measures the significance of each regression coefficient and orders them accordingly. In other words, the most important features will correspond to the ones with the most significant p-values.

Feature interaction

Greenwell et al. (2018)[1] also proposed to use the PD to measure the feature interaction of two given features. Let $SD(X_i, X_j)$, with $i \neq j$, be the standard deviation of the joint PD values $f_{ij}(X_{ii'}, X_{jj'})$, for $i' = 1, 2, \dots, k_i$ and $j' = 1, 2, \dots, k_j$. The intuition proposed by the authors is that a weak interaction effect between X_i and X_j on the response Y would suggest that importance $i(X_i, X_j)$ has little variance when either X_i or X_j is held constant and the other varies.

The computation of the feature interaction is straightforward. Consider any two features (X_i, X_j) , $i \neq j$. We construct the PD function $f_{ij}(X_i, X_j)$ and compute the feature importance of X_i while keeping X_j constant, for all values of X_j . We denote it by $SD(X_i|X_j)$. Following that, we take the standard deviation of the resulting importance scores across all values of X_j . We denote the latter quantity as $i(X_i|X_j)$. Similarly, we compute $i(X_j|X_i)$. To compute the feature interaction, one simply averages the two results. A large values will indicate a possible feature interaction.

Although the results reported by Greenwell et al. (2018)[1] seem encouraging, the authors do not offer a rigorous justification for their proposal which makes the method to appear rather heuristic. In the following paragraphs, we provide through concrete examples some arguments on why the method can capture feature interactions and which are some failure cases.

Consider the following function of two variables, $f : [0, 1] \rightarrow \mathbb{R}$, $f(X_1, X_2) = X_1 + X_2 + X_1X_2$. Due to its simplistic form, one might be tempted to eyeball the decomposition of the function in three terms: a main effect of X_1 given by $f_1(X_1)$, a main effect of X_2 given by $f_2(X_2)$, and an interaction term between (X_1, X_2) given by $f_3(X_1, X_2) = X_1X_2$. Although this can be a valid function decomposition within an axiomatic framework, it is not the case for the PD. This is because in the PD case the term X_1X_2 does not only contain a feature interaction between X_1 and X_2 , but also contains a fraction of their main effects.

To understand why X_1X_2 contains also a fraction of the main effects of X_1 and X_2 , we first provide an intuitive view of what the main effect consists when using the PD functions. Informally, one can think of the main effect of a feature in the PD context as how well w.r.t. the mean squared error (MSE) one can approximate $f(X_1, X_2)$ by only having access to the feature X_1 . In our case we can analyze the three terms:

- $f_1(X_1) = X_1$ is straightforward. Since we have access to the feature X_1 , we can approximate the function exactly.
- $f_2(X_2) = X_2$ is also relatively easy. Since we only have access to X_1 and because $f_2(X_2)$ does not depend on X_1 , the best we can do is to approximate it with a constant. The constant that we choose is dependent of the objective we want to minimize. For MSE, the constant is given by $\mathbb{E}[f_2(X_2)]$, where the expectation is taken w.r.t. the marginal of X_2 .
- $f_3(X_1, X_2) = X_1X_2$ is a bit more challenging. As mentioned before, one might be tempted to say that f_3 describes only the feature interaction between X_1 and X_2 , but given our PD approach, this is not the case. This is because in the PD context, the f_3 contains also a fractions of the main effects of X_1 and X_2 .

We now elaborate more on the third bullet point. To intuitively see why this is the case, let us fix the value $X_1 = 0.2$ and inspect $f_3(0.2, X_2)$ by varying X_2 (see Figure 7(a)).

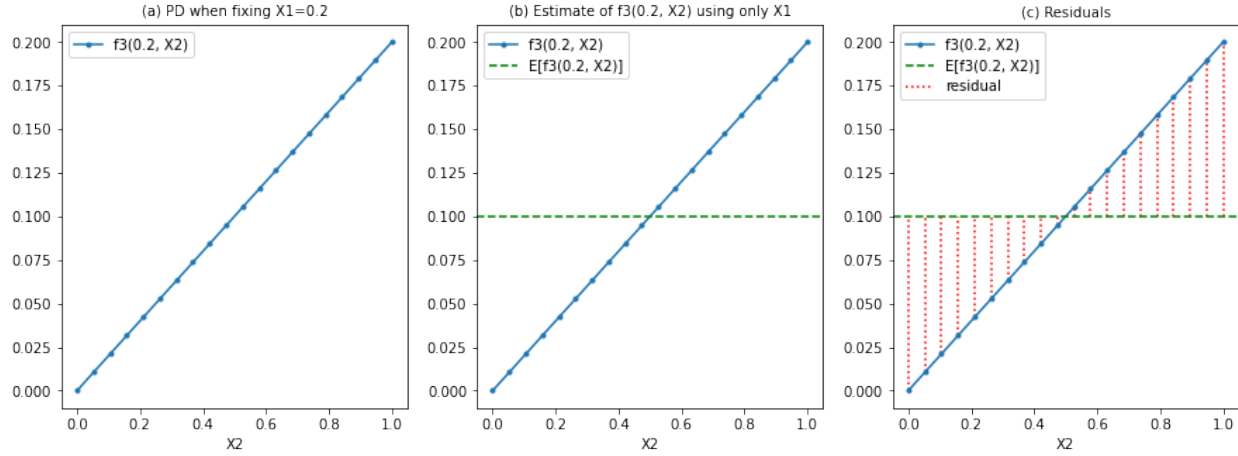


Figure 7. Conditional PD estimation steps.

Having no access to X_2 , the best we can predict $f_3(X_1, X_2)$ based only on $X_1 = 0.2$ w.r.t. the MSE, is to predict the average response of $f_3(0.2, X_2)$ over the marginal of X_2 . Formally, the value is given by $\mathbb{E}_{X_2}[f_3(0.2, X_2)]$. This is depicted by the green line in Figure 7(b).

The residuals (i.e. what we cannot predict) are given by $f_3(0.2, X_2) - \mathbb{E}[f_3(0.2, X_2)]$, displayed in red in Figure 7(c).

The residuals/errors can be attributed to the following:

- either to a fraction from the main effect of X_2 .
- either to the interaction between X_1 and X_2 .
- or a combination of both.

Intuitively, we would conclude that we do not have any feature interaction between X_1 and X_2 if for any value of $X_1 = c$, we obtain the same error patterns. In this case, the errors would only be a result from a fraction of the main effect of X_2 . Figure 8 displays the two scenarios, without and with feature interaction.

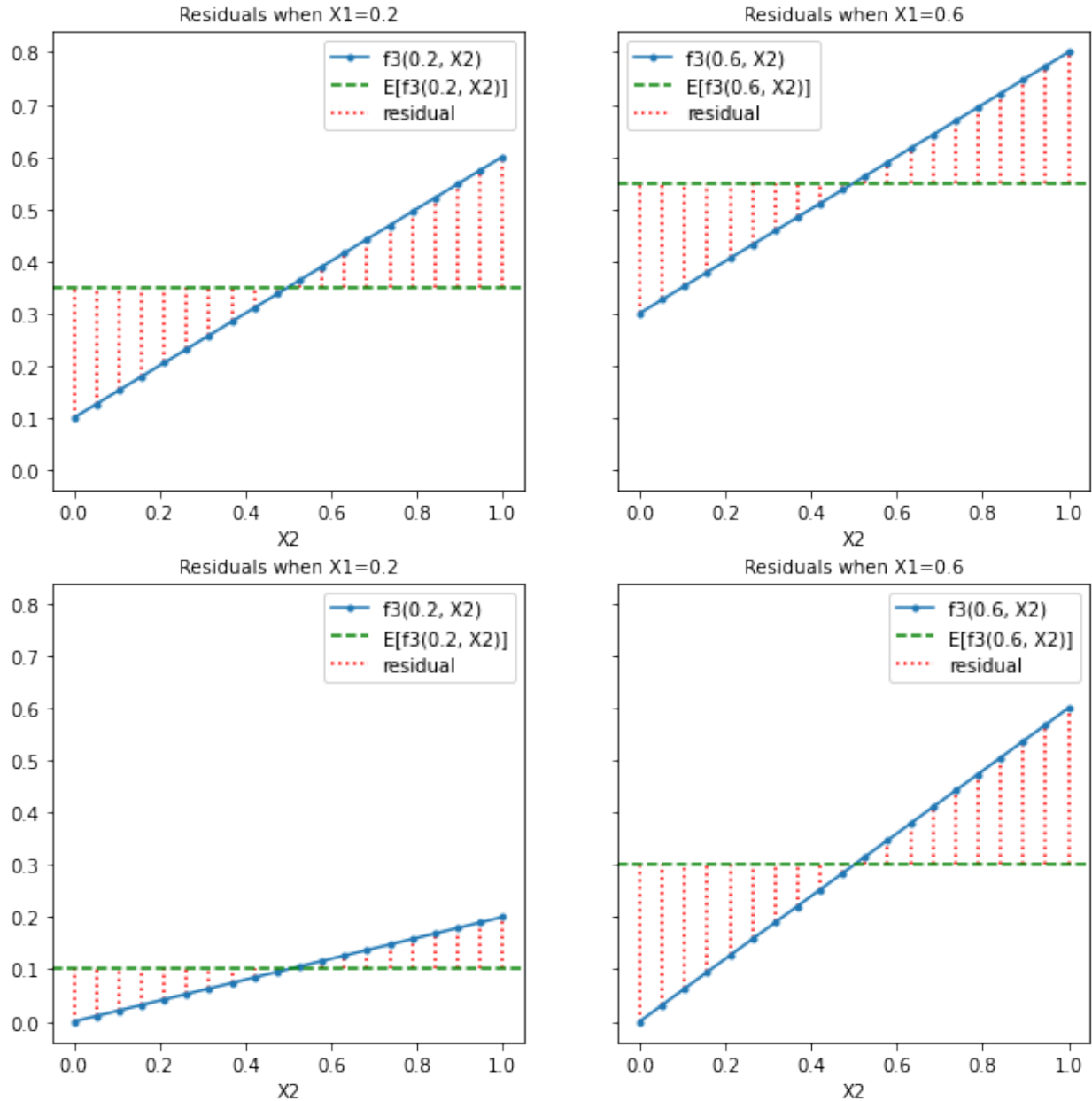


Figure 8. Conditional PD estimation error patterns. The first row displays error patterns when there is no interaction (i.e., the interaction term is removed). The second row displays error pattern when there is interaction.

We now come back to [Greenwell et al. \(2018\)\[1\]](#) proposal on how to measure the feature interaction. For feature X_1 , the first step is to compute the standard deviation of the PD along the X_2 axis when X_1 is held constant. This is equivalent of computing the root mean squared error (i.e., corresponds to the red lines above). This first step can quantify whether there is some effect we are missing just by using X_1 , which we can attribute to either a fraction from the main effect of X_2 or to the interaction between X_1 and X_2 (we still do not know to which one we should attribute). We repeat the same step for every value of X_1 . Having all the standard deviations for every value of X_{1i} (i.e., $SD_{X_2}(X_2|X_{1i})$), we compute the standard deviation along the X_1 , $i(X_1|X_2) = SD_{X_1}[SD_{X_2}(X_2|X_1)]$. This step is necessary (but not sufficient) to check whether the variance in the error is attributed to the fraction from the main effect of X_2 or to the feature interaction between X_1 and X_2 . Note that if the standard deviation $i(X_1|X_2) = SD_{X_1}[SD_{X_2}(X_2|X_1)] = 0$, it means that we have the same variance along the X_2 for every value X_{1i} . This might happen for multiple reasons, but one reason can be if we encounter the same error pattern when we condition on every X_{1i} . If that happens then it

means that there is no feature interaction and all the error is attributed to a fraction of the main effect of X_2 .

One simple example for which the method fails to capture the feature interaction is for $f : [-1, 1] \times [-1, 1] \rightarrow \mathbb{R}$, $f(X_1, X_2) = \mathbb{I}(X_1 X_2 > 0)$. In this case the variance is the same when keeping X_1 or X_2 constant, but the error patterns differ depending on their sign. The PD and the conditional importances are displayed in Figure 9:

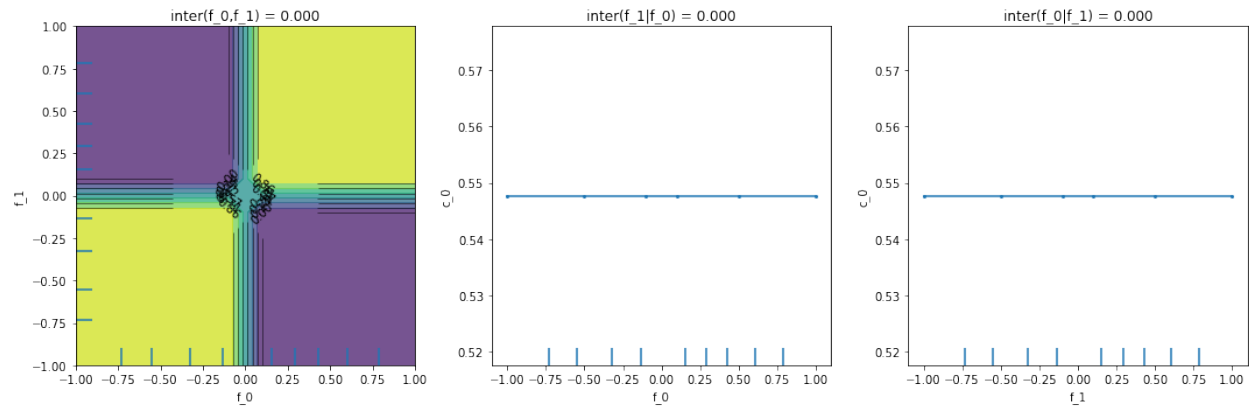


Figure 9. Feature interaction failure case.

Although there is clear interaction between X_1 and X_2 , the method fails to detect it because the variance along the X_1 and X_2 axis is the same.

7.10.4 Examples

Partial Dependence Variance, regression example (Friedman’s regression problem)

7.10.5 References

- [1] Greenwell, Brandon M., Bradley C. Boehmke, and Andrew J. McCarthy. “A simple and effective model-based variable importance measure.” arXiv preprint arXiv:1805.04755 (2018).
- [2] Friedman, Jerome H. “Greedy function approximation: a gradient boosting machine.” *Annals of statistics* (2001): 1189-1232.
- [3] Molnar, Christoph. *Interpretable machine learning*. Lulu. com, 2020.
- [4] Fanaee-T, Hadi, and Gama, Joao, ‘Event labeling combining ensemble detectors and background knowledge’, *Progress in Artificial Intelligence* (2013): pp. 1-15, Springer Berlin Heidelberg.
- [5] Breiman, Leo. “Random forests.” *Machine learning* 45.1 (2001): 5-32.
- [6] Friedman, Jerome H. “Greedy function approximation: a gradient boosting machine.” *Annals of statistics* (2001): 1189-1232.

[\[source\]](#)

7.11 Permutation Importance

7.11.1 Overview

The permutation importance, initially proposed by Breiman (2001)[1], and further refined by Fisher et al. (2019)[2] is a method to compute the global importance of a feature for a tabular dataset. The computation of the feature importance is based on how much the model performance degrades when the feature values within a feature column are permuted. By inspecting the attribution received by each feature, a practitioner can understand which are the most important features that the model relies on to compute its predictions.

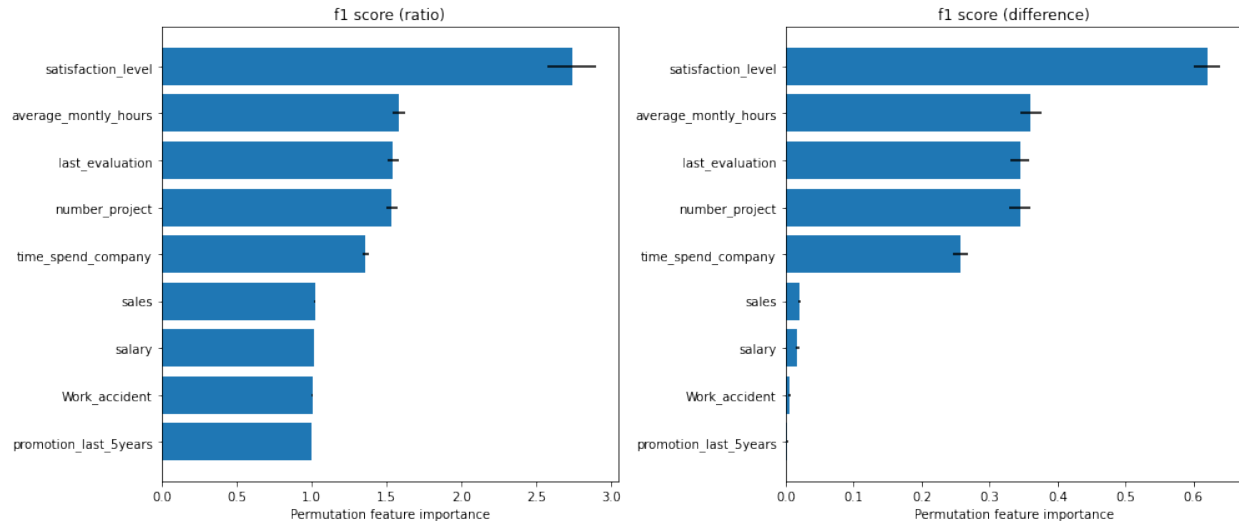


Figure 1. Permutation Importance using F_1 score on “Who’s Going to Leave Next?” dataset. Left figure displays the importance as the ratio between the original score and the permuted score. Right figure displays the importance as the difference between the original score and the permuted score.

Figure 1 displays the importance of each feature according to the F_1 score function reported as the ratio between the original score and the permuted score (left plot), and as the difference between the original score and the permuted score (right plot). We can observe that the most important feature that the model relies on is the `satisfaction_level`. Following that, we have three features that have approximately the same importance, namely the `average_monthly_hours`, `last_evaluation` and `number_project`. Finally, in our top 5 hierarchy we have `time_spend_company`. Features like `sales`, `salary`, `Work_accident` and `promotion_last_5years` receive an importance close to 1 in the left plot and an importance close to 0 in the right plot which are an indication that the features are not important to the model. For a more detailed analysis, please check the worked [example](#).

For pros & cons, see the [Permutation Importance](#) section from the [Introduction](#) materials.

7.11.2 Usage

To initialize the explainer with any black-box model, one can directly pass the prediction function, the metrics consisting of the loss functions or the score functions, and optionally a list of feature names:

```
from alibi.explainers import PermutationImportance

pfi = PermutationImportance(predictor=predict_fn,
                             loss_fns=loss_fns,
                             score_fns=score_fns,
                             feature_names=feature_names)
```

Note

Remember that the `PermutationImportance` explainer measures the importance of a feature `f` as the degradation of the model when the feature values of `f` are permuted. The degradation of the model can thus be quantified as either the increase in the loss function or the decrease in the score function. Although one can transform a loss function into a score function and vice-versa (i.e., simply negate the value and optionally add an offset), the equivalent representation might not always be natural to interpret (e.g., transforming mean squared error loss into the equivalent score given by the negative mean squared error). Thus, the `alibi` API allows the user to provide the suitable metric either as a loss or a score function.

The metric (loss or score) functions can be initialized through strings, a callable or dictionaries. For example, for a classification problem, the initialization of the score functions through strings can be done either `score_fns = ['accuracy', 'f1']` or directly `score_fns='f1'` when a single score function is used. Similarly, when a single score function is used the initialization through a callable can be done as `score_fns = accuracy_score`, where `accuracy_score` is the reference to the function. Finally, the initialization through a dictionary would look like `score_fns={'name_score_1': function_score_1, 'name_score_2': function_score_2}`. For all the previous cases, the initialization is analogous for the loss functions.

Note that the initialization through a callable or a dictionary allows the flexibility to provide custom metric functions. The signature of a metric function must be as follows:

```
def metric_fn(y_true: np.ndarray,
              y_pred: np.ndarray,
              sample_weight: Optional[np.ndarray] = None) -> float:
    pass
```

or

```
def metric_fn(y_true: np.ndarray,
              y_score: np.ndarray,
              sample_weight: Optional[np.ndarray] = None) -> float:
    pass
```

where `y_true` is the array of ground-truth values, `y_pred` | `y_score` is the output of the predictor used in the initialization of the explainer, and `sample_weight` is an optional array containing the weights for the given data instances.

Besides designing custom metrics, the signature above makes it possible to use the `sklearn` metrics provided [here](#). Also, the list of all supported string metrics can be found [here](#).

Following the initialization, we can produce an explanation given the test dataset ($X_{\text{test}}, y_{\text{test}}$):

```
exp = pfi.explain(X=X_test, y=y_test)
```

Multiple arguments can be provided to the `explain` method:

- `X` - A $N \times F$ input feature dataset used to calculate the permutation feature importance. This is typically the test dataset.
- `y` - Ground-truth labels array of size N (i.e. $(N,)$) corresponding the input feature `X`.
- `features` - An optional list of features or tuples of features for which to compute the permutation feature importance. If not provided, the permutation feature importance will be computed for every single features in the dataset. Some example of `features` would be: `[0, 2]`, `[0, 2, (0, 2)]`, `[(0, 2)]`, where `0` and `2` correspond to column `0` and `2` in `X`, respectively.

- **method** - The method to be used to compute the feature importance. If set to 'exact', a “switch” operation is performed across all observed pairs, by excluding pairings that are actually observed in the original dataset. This operation is quadratic in the number of samples ($N \times (N - 1)$ samples) and thus can be computationally intensive. If set to 'estimate', the dataset will be divided in half. The values of the first half containing the ground-truth labels the rest of the features (i.e. features that are left intact) is matched with the values of the second half of the permuted features, and the other way around. This method is computationally lighter and provides estimate error bars given by the standard deviation. Note that for some specific loss and score functions, the estimate does not converge to the exact metric value.
- **kind** - Whether to report the importance as the loss/score ratio or the loss/score difference. Available values are: 'ratio' | 'difference'.
- **n_repeats** - Number of times to permute the feature values. Considered only when `method='estimate'`.
- **sample_weight** - Optional weight for each sample instance.

Note

As mentioned in the parameter description, depending on the loss or score functions used to measure the model performance, the feature importance values when using `method='estimate'` might not converge to the feature importance values when `method='exact'`, regardless of the number of times the feature values are permuted specified via `n_repeats`.

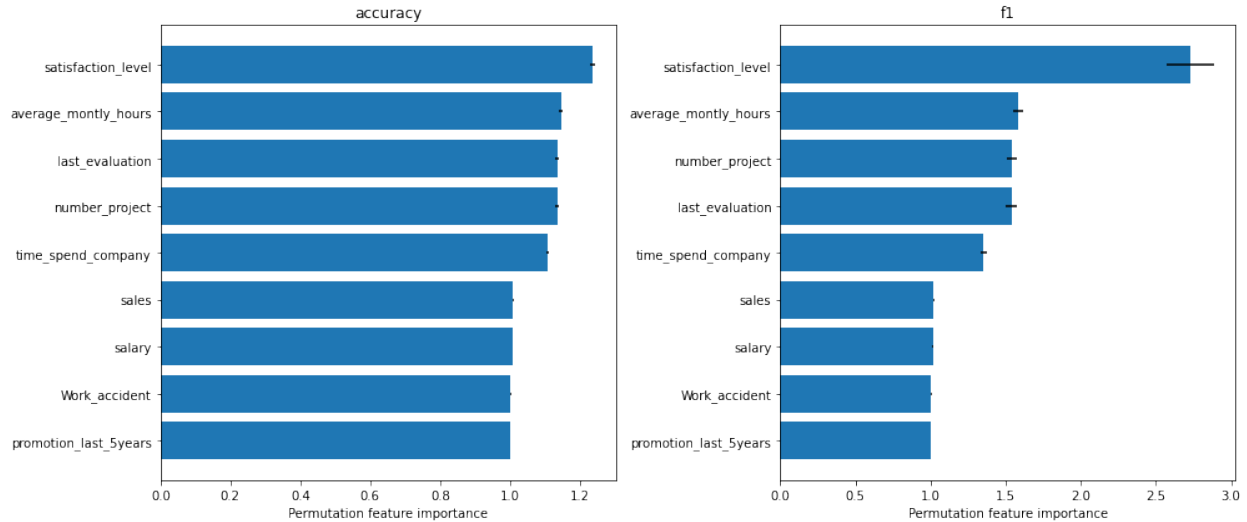
The result `exp` is an `Explanation` object which contains the following data-related attributes:

- **feature_names** - A list of strings or tuples of strings containing the names associated with the explained features.
- **metric_names** - A list of strings containing the names of the metrics used to compute the feature importance.
- **feature_importance** - A list of lists of float when `method='exact'` and list of lists of dictionary when `method='estimate'` containing the feature importance for each metric and each explained feature. When `method='estimate'`, the dictionary returned for each metric and each feature contains the importance mean, the importance standard deviation and the samples used to compute those statistics.

For convenience, we included a plotting function `plot_permutation_importance` which produces a bar plot with the feature importance values for each metric using `matplotlib`.

```
from alibi.explainers import plot_permutation_importance
plot_permutation_importance(exp)
```

The following figure displays the feature importance for the accuracy and F_1 score for a random forest classifier trained on the “Who’s Going to Leave Next?” dataset (see worked [example](#)).



7.11.3 Theoretical exposition

Breiman (2001)[1] initially proposed the permutation feature importance for a random forest classifier as a method to compute the global importance of a feature as seen by the model. More precisely, consider a dataset with M input features and a random forest classifier. After each tree is created, the values of the m -th feature in the out-of-bag (OOB) split are randomly permuted and the newly generated data is fed to the current tree to obtain a new prediction. The result for each newly generated data instance from OOB is saved. The process is then repeated for all features $m = 1, 2, \dots, M$. After the procedure is completed for every tree, the noised responses are compared with the true label to give the misclassification rate. The importance of each feature is given by the percent increase in the misclassification rate as compared with the OOB rate when all the features are left intact.

Note

The intuition behind the procedure described above is that an increase in the misclassification rate is an indication that a feature is important for the given model.

Although the method was initially proposed for a random forest classifier, it can be easily generalized to any model and prediction task (e.g., classification or regression). Fisher et al. (2019)[2] proposed a model agnostic version of the permutation feature importance called *model reliance* which is the one implemented in alibi.

Notation

Before diving into the mathematical formulation of the model reliance, we first introduce some notation. Let $Z = (Y, X_1, X_2) \in \mathcal{Z}$ be an *iid* random variable with outcome $Y \in \mathcal{Y}$ and covariates (features) $X = (X_1, X_2) \in \mathcal{X}$, where the covariates subsets $X_1 \in \mathcal{X}_1$ and $X_2 \in \mathcal{X}_2$ may be each multivariate. The goal is to measure how much the model prediction relies on X_1 to predict Y .

For a given prediction model f , Fisher et al. (2019)[2] introduced the *model reliance* to be the percent increase in f 's expected loss when noise is added to X_1 . Informally this can be written as:

$$MR(f) = \frac{\text{Expected loss of } f \text{ under noise}}{\text{Expected loss of } f \text{ without noise}}$$

Note that there are certain properties that the noise must satisfy:

- must render X_1 completely uninformative of the outcome Y .

- must not alter the marginal distribution of X_1 .

Definition

Given the notation above, we can introduce formally the *model reliance*.

Let $Z^{(a)} = (Y^{(a)}, X_1^{(a)}, X_2^{(b)})$ and $Z^{(b)} = (Y^{(b)}, X_1^{(b)}, X_2^{(b)})$ be independent random variables, each following the same distribution as $Z = (Y, X_1, X_2)$. The expected loss of the model f across pairs of observations $(Z^{(a)}, Z^{(b)})$ in which the values $X_1^{(a)}$ and $X_1^{(b)}$ have been switched is defined as:

$$e_{\text{switch}}(f) = \mathbb{E}[L\{f, (Y^{(b)}, X_1^{(a)}, X_2^{(b)})\}]$$

Note that the definition above uses the pair $(Y^{(b)}, X_2^{(b)})$ from $Z^{(b)}$, but the variable $X_1^{(a)}$ from $Z^{(a)}$, hence the name *switched*. It is important to understand that the values $(Y^{(b)}, X_1^{(a)}, X_2^{(b)})$ do not relate to each other and thus we brake the correlation between X_1 with the remaining features X_2 and with the output Y . An alternative interpretation of $e_{\text{switch}}(f)$ is the expected loss of f when noise is added to X_1 in such a way that X_1 becomes completely uninformative of Y , but the marginal of X_1 is unchanged.

The reference quantity to compare $e_{\text{switch}}(f)$ against is the standard expected loss when the features are left intact (i.e., none of the feature values were switched). Formally it can be written as:

$$e_{\text{orig}}(f) = \mathbb{E}[L\{f, (Y, X_1, X_2)\}]$$

Given the two quantities above, we can formally define $MR(f)$ as their ratio:

$$MR(f) = \frac{e_{\text{switch}}(f)}{e_{\text{orig}}(f)}$$

There are three possible cases to be analyzed:

- $MR(f) > 1$ indicates that the model f relies on the feature X_1 . For example, a $MR(f) = 2$ means that the error loss has doubled when X_1 was permuted.
- $MR(f) = 1$ indicates that the model f **does not** rely on the feature X_1 . This means that the error has not changed when X_1 was permuted.
- $MR(f) < 1$ is an interesting case. Surprisingly, there exist models f such that their reliance is less than one. For example, this can happen if the model f treats X_1 and Y as positively correlated when in fact they are negatively correlated. In many cases, a $MR(f) < 1$ implies the existence of a better performant model f' satisfying $MR(f') = 1$ and $e_{\text{orig}}(f') < e_{\text{orig}}(f)$. This is equivalent to saying that the model f is typically suboptimal.

An alternative definition of the model reliance which uses the difference instead of the ratio is given by:

$$MR_{\text{difference}}(f) = e_{\text{switch}}(f) - e_{\text{orig}}(f).$$

As emphasized by [Molnar 2020\[3\]](#), the positive aspect of using ratio over difference is that the results are comparable across multiple problems.

Estimation of model reliance with U-statistics

For a given model f and a dataset $Z = (Y, X_1, X_2)$, one has to estimate the $MR(f)$. The estimation of the $e_{\text{orig}}(f)$ is straightforward through the empirical loss, formally given by:

$$\hat{e}_{\text{orig}}(f) = \frac{1}{n} \sum_{i=1}^n L\{f, Y^{(i)}, X_1^{(i)}, X_2^{(i)}\}.$$

For the estimation of the $e_{\text{switch}}(f)$, one has to be more considerate because applying a naive permutation of the feature values can be a source of bias. To be more concrete on how the bias can be introduced, let us consider an example of four data instances

$$\mathcal{Z} = \{(Y^{(1)}, X_1^{(1)}, X_2^{(1)}), (Y^{(2)}, X_1^{(2)}, X_2^{(2)}), (Y^{(3)}, X_1^{(3)}, X_2^{(3)}), (Y^{(4)}, X_1^{(4)}, X_2^{(4)})\}.$$

Note that naively applying the permutation (1, 2, 4, 3) to the original dataset will only break the correlation for two instances out of four, and the rest will be left intact. Since the first two instances will be left intact and since they follow the same data distribution that the model was trained on, we expect that the error for those instances to be low (i.e., if we use the test set and if the model did not overfit), which will bring down the estimate of $e_{\text{switch}}(f)$. Thus, permutations π for which there exist elements such that $\phi(i) = i$ are a source of bias in our estimate.

Fisher et al. (2019)[2] proposed two alternative methods to compute an unbiased estimate using U-statistic. The first estimate is to perform a “switch” operation across all observed pairs, by excluding pairings that are actually observed in the original dataset. Formally, it can be written as:

$$\hat{e}_{\text{switch}}(f) = \frac{1}{n(n-1)} \sum_{i=1}^n \sum_{j \neq i} L\{f, (Y^{(i)}, X_1^{(i)}, X_2^{(i)})\}.$$

The computation of the $\hat{e}_{\text{switch}}(f)$ can be expensive because the summation is performed over all $n(n-1)$ possible pairs.

If the estimation is prohibited due to the sample size, the following alternative estimator can be used:

$$\hat{e}_{\text{divide}}(f) = \frac{1}{2\lfloor n/2 \rfloor} \sum_{i=1}^{\lfloor n/2 \rfloor} [L\{f, (Y^{(i)}, X_1^{(i+\lfloor n/2 \rfloor)}, X_2^{(i)})\} + L\{f, (Y^{(i+\lfloor n/2 \rfloor)}, X_1^{(i)}, X_2^{(i+\lfloor n/2 \rfloor)})\}].$$

Note that rather than summing over all possible pairs, the dataset is divided in half and the first and half values for (Y, X_2) are matched with the second half values of X_1 , and the other way around. Besides the light computation, this approach can provide confidence intervals by computing the estimates over multiple data splits.

We end our theoretical exposition by mentioning that both estimators above can be used to compute an unbiased estimate of $\hat{M}R(f)$. Furthermore, one interesting observation is that the definition of e_{switch} is very similar to the one proposed by Breiman (2001)[1]. Formally, the approach described by Breiman (2001)[1] can be written as:

$$\hat{e}_{\text{permute}} = \sum_{i=1}^n L\{f, (Y^{(i)}, X_1^{\pi_l(i)}, X_2^{\pi_l(i)})\},$$

where $\pi_j \in \{\pi_1, \dots, \pi_{n!}\}$ is one permutation from the set of all permutations of $(1, \dots, n)$. The calculation proposed by Fisher et al. (2019)[2] is proportional to the sum of losses over all $n!$ permutations, excluding the n unique combinations of the rows of X_1 and the rows of $[Y, X_2]$ that appear in the original sample. As mentioned before, excluding those combinations is necessary to preserve the unbiasedness of the $\hat{e}_{\text{switch}}(f)$.

7.11.4 Examples

Permutation Importance classification example (“Who’s Going to Leave Next?”)

7.11.5 References


- [1] Breiman, Leo. “Random forests.” Machine learning 45.1 (2001): 5-32.
 - [2] Fisher, Aaron, Cynthia Rudin, and Francesca Dominici. “All Models are Wrong, but Many are Useful: Learning a Variable’s Importance by Studying an Entire Class of Prediction Models Simultaneously.” J. Mach. Learn. Res. 20.177 (2019): 1-81.
 - [3] Molnar, Christoph. Interpretable machine learning. Lulu. com, 2020.
- [source]

7.12 Similarity explanations

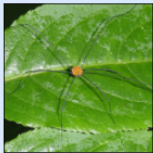
7.12.1 Overview

The `GradientSimilarity` class implements an explanation method that belongs to the family of the [similarity-based explanations](#) methods.

Given an input instance of a machine learning model, similarity-based methods aim to explain the output of the model by finding and presenting instances seen during training that are similar to the given instance. Roughly speaking, an explanation of this type should be interpreted by the user following a rationale of the type: *This X is a Y because a similar instance X' is a Y .*

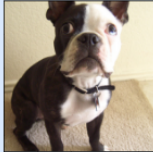


is a **Boston Terrier** because




is a **harvestman** because

a similar instance



is a **Boston Terrier**.

a similar instance



is a **harvestman**.

"I really do **feel** as if I have been **neglectful** in my duties here at the Cyber Homestead." expresses **sadness** because

a similar sentence "**I feel neglectful** and regret but my patient friend I'm coming soon to read what posts you send." expresses **sadness**.

"I **feel free** like I could do whatever I imagined everything I ever wished I could do infinite." expresses **joy** because

a similar sentence "I was **feeling free** and decided to just run on feel and hope for the best." expresses **joy**.

Similarity explanations of a ResNet50 model on ImageNet dataset (top) and of a DistilBERT model on Emotions dataset (bottom).

Various similarity-based methods use different metrics to quantify the similarity between two instances. The `GradientSimilarity` class implements gradients-based metrics, as introduced by [Charpiat et al., 2019](#).

Theory

The purpose of gradient-based methods is to define a similarity kernel between two instances that quantify how similar the instances are *according to a model* trained for a specific task (for example a classifier). In particular, given two instances $z = (x, y)$ and $z' = (x', y')$, a model $f_\theta(x)$ parametrized by θ and a loss function $\mathcal{L}_\theta(z) = \mathcal{L}(f_\theta(x), y)$, we define similarity as the influence of z over z' with respect to the loss function. The similarity quantifies how much an additional parameter's update that changes the loss calculated at z by a certain amount would change the loss calculated at z' .

In particular, let us consider the Taylor expansion of the loss function \mathcal{L} at the point z , which reads like:

$$\mathcal{L}_{\theta+\delta\theta}(z) = \mathcal{L}_\theta(z) + \delta\theta \nabla_\theta \mathcal{L}_\theta(z) + \mathcal{O}(\|\delta\theta\|^\epsilon)$$

If we want to change the loss at z by an amount ϵ , we can do so by changing the model's parameters θ by an amount $\delta\theta = \epsilon \frac{\nabla_\theta \mathcal{L}_\theta(z)}{\|\nabla_\theta \mathcal{L}_\theta(z)\|^2}$. In fact, by substituting this value in the Taylor expansion above we obtain:

$$\mathcal{L}_{\theta+\delta\theta}(z) = \mathcal{L}_\theta(z) + \epsilon + \mathcal{O}(|\epsilon|^2)$$

Now, we would like to measure the impact of such a change of parameters on the loss function calculated at a different point z' . Using Taylor expansion again, the loss at point z' is given by:

$$\mathcal{L}_{\theta+\delta\theta}(z') = \mathcal{L}_\theta(z') + \delta\theta \nabla_\theta \mathcal{L}_\theta(z') + \mathcal{O}(\|\delta\theta\|^\epsilon)$$

Substituting $\delta\theta = \epsilon \frac{\nabla_\theta \mathcal{L}_\theta(z)}{\|\nabla_\theta \mathcal{L}_\theta(z)\|^2}$ we have

$$\mathcal{L}_{\theta+\delta\theta}(z') = \mathcal{L}_\theta(z') + \epsilon \frac{\nabla_\theta \mathcal{L}_\theta(z') \cdot \nabla_\theta \mathcal{L}_\theta(z)}{\|\nabla_\theta \mathcal{L}_\theta(z)\|^2} + \mathcal{O}(\|\epsilon\|^\epsilon).$$

In conclusion, the kernel

$$k_\theta(z, z') = \frac{\nabla_\theta \mathcal{L}_\theta(z') \cdot \nabla_\theta \mathcal{L}_\theta(z)}{\|\nabla_\theta \mathcal{L}_\theta(z)\|^2}$$

quantifies how much the loss function at point z' has changed after a parameters' update that has changed the loss at point z by an amount ϵ . It represents the influence that the point z has over the point z' with respect to the loss function.

Based on this kernel, which is not symmetric, the original paper suggests two symmetric alternatives:

$$k_\theta(z, z') = \frac{\nabla_\theta \mathcal{L}_\theta(z') \cdot \nabla_\theta \mathcal{L}_\theta(z)}{\|\nabla_\theta \mathcal{L}_\theta(z')\| \|\nabla_\theta \mathcal{L}_\theta(z)\|}.$$

$$k_\theta(z, z') = \nabla_\theta \mathcal{L}_\theta(z') \cdot \nabla_\theta \mathcal{L}_\theta(z).$$

All the three versions of the kernel are implemented in the `GradientSimilarity` class (see [Usage](#) section below).

7.12.2 Usage

Initialization


```

from alibi.explainers.similarity.grad import GradientSimilarity

model = <YOUR_MODEL>
loss_fn = <YOUR_LOSS_FUNCTION>

explainer = GradientSimilarity(predictor=model, # your tensorflow or pytorch model.
                              loss_fn=loss_fn, # your loss_fn. Usually the loss_
↪function of your model.
                              sim_fn='grad_dot', # 'grad_dot', 'grad_cos' or 'grad_asym_
↪dot'.
                              task='classification', # 'classification' or 'regression'.
                              precompute_grads=False, # precompute training set_
↪gradients in fit step.
                              backend='tensorflow', # 'tensorflow' or 'pytorch'.
                              device=None, # pytorch device. For example 'cpu' or 'cuda'.
                              verbose=False)

```

- **predictor**: The GradientSimilarity class provides both a tensorflow and a pytorch backend, so your predictor can be a model in either of these frameworks. The backend argument must be set accordingly.
- **loss_fn**: The loss function $\mathcal{L}(f_{\theta}(x), y)$ used to compute the gradients. Usually the loss function used by the model for training, but it can be any function taking as inputs the model's prediction and the labels y -s.
- **sim_fn**: The similarity function used to compute the kernel $k_{\theta}(z, z')$. GradientSimilarity implements 3 kernels:
 - ‘grad_dot’, defined as

$$k_{\theta}(z, z') = \nabla_{\theta} \mathcal{L}_{\theta}(z') \cdot \nabla_{\theta} \mathcal{L}_{\theta}(z).$$

- ‘grad_cos’, defined as

$$k_{\theta}(z, z') = \frac{\nabla_{\theta} \mathcal{L}_{\theta}(z') \cdot \nabla_{\theta} \mathcal{L}_{\theta}(z)}{\|\nabla_{\theta} \mathcal{L}_{\theta}(z')\| \|\nabla_{\theta} \mathcal{L}_{\theta}(z)\|}.$$

- ‘grad_asym_dot’, defined as

$$k_{\theta}(z, z') = \frac{\nabla_{\theta} \mathcal{L}_{\theta}(z') \cdot \nabla_{\theta} \mathcal{L}_{\theta}(z)}{\|\nabla_{\theta} \mathcal{L}_{\theta}(z)\|^2}.$$

- **precompute_grads**: Whether to pre-compute the training set gradients during the fit step or not.
- **backend**: Backend framework. tensorflow or pytorch.
- **device**: pytorch device. For example cpu or cuda.

Fit

Fitting is straightforward, just passing the training set:

```
explainer.fit(X_train, y_train)
```

In this step, the dataset and the data input dimensions are stored as attributes of the class. If **precompute_grads=True**, the gradients for all the training instances are computed and stored as attributes.

Explanation

We can now explain the instance by running:

```
explanation = explainer.explain(X, y)
```

- **X**: Test instances to be explained.
- **y**: Target class (optional). This array can contain either a single entrance that is applied for all test instances or multiple entrances, one for each test instance.

The returned explanation is a standard `alibi` explanation class with the following data attributes:

- **scores**: A numpy array with the similarity score for each train instance.
- **ordered_indices**: A numpy array with the indices corresponding to the train instances, ordered from the most similar to the least similar.
- **most_similar**: A numpy array with the 5 most similar instances in the train set.
- **least_similar**: A numpy array with the 5 least similar instances in the train set.

Notes on usage

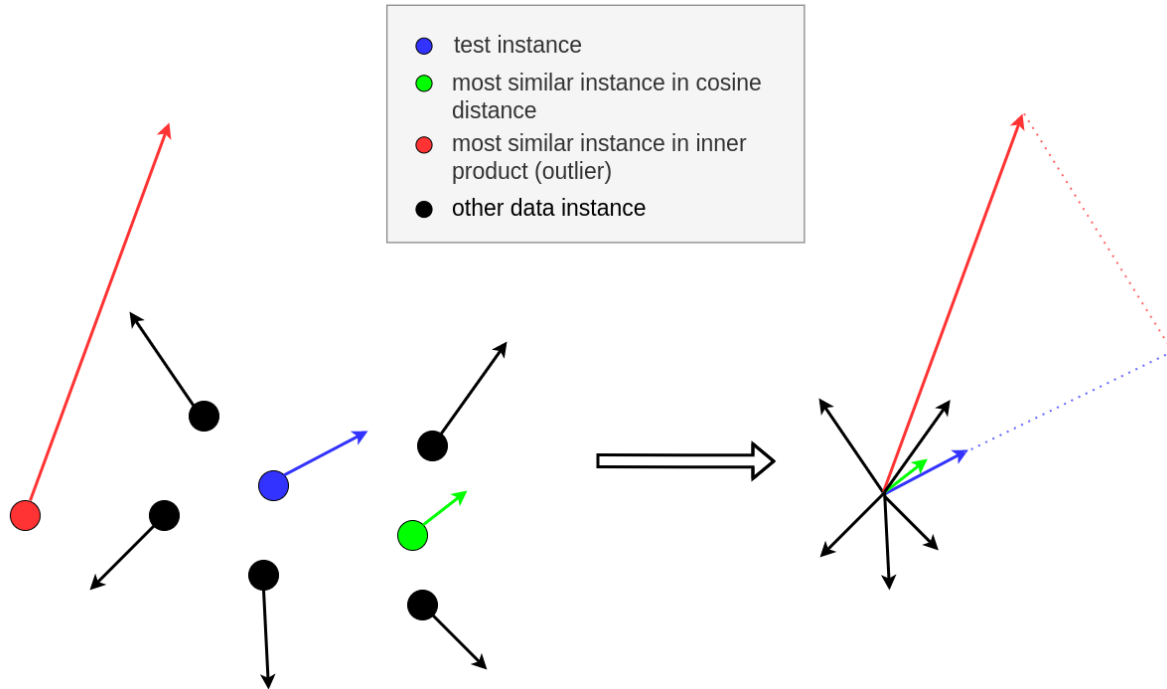
Fitting and train set

The `GradientSimilarity` will order the instances passed on the fit step based on the similarity with the instances passed on the explain step, regardless of whether they have been used for training the model or not. In the [examples below](#) we downsample the training set by picking a number of random instances in order to speed up the fit step.

Setting `precompute_grads=True` will speed up the computation during the explain step, but the fit step will require considerably more time as the gradients for all the training instances are computed. It could also require a considerable amount of memory for large datasets as all the gradients are stored as attributes of the class instance.

Similarity metrics

As reported in [Hanawa et al. \(2021\)](#), the `grad_dot` metrics fails the identical class test, meaning that not always the most similar instances produced belong to the same class of the instance of interest. On the other hand, it is highly likely that the most similar instances belong to the same class as the instance of interest when the `grad_cos` metric is used. Note that an important feature of the cosine distance is the normalization coefficient which makes the method insensitive to outliers (i.e. instances with large gradient norms) as illustrated in the following figure:



Left: 2D data instances (circles) and their corresponding gradients (arrows) for a model parametrized by two parameters; Right: gradient comparison.

Batch explanations

When a batch of instances is passed to explain, a naive loop over the instances is performed internally and the gradients are calculated one instance at a time. This is due to limitations in the `tensorflow` and `pytorch` backends which automatically aggregate the values of the gradients in a batch.

7.12.3 Examples

Similarity explanation on MNIST

Similarity explanation on ImageNet

Similarity explanation on 20 news groups

[source]

7.13 Tree SHAP

Note

To enable SHAP support, you may need to run:

```
pip install alibi[shap]
```

7.13.1 Overview

The tree SHAP (SHapley Additive exPlanations) algorithm is based on the paper [From local explanations to global understanding with explainable AI for trees](#) by Lundberg et al. and builds on the open source [shap library](#) from the paper's first author.

The algorithm provides human interpretable explanations suitable for regression and classification of models with tree structure applied to tabular data. This method is a member of the *additive feature attribution methods* class; feature attribution refers to the fact that the change of an outcome to be explained (e.g., a class probability in a classification problem) with respect to a *baseline* (e.g., average prediction probability for that class in the training set) can be attributed in different proportions to the model input features.

A simple illustration of the explanation process is shown in Figure 1. Here we see depicted a tree-based model which takes as an input features such as Age, BMI or Blood pressure and outputs Mortality risk score, a continuous value. Let's assume that we aim to explain the difference between an observed outcome and no risk, corresponding to a base value of 0.0. Using the Tree SHAP algorithm, we attribute the 4.0 difference to the input features. Because the sum of the attribute values equals output - base value, this method is *additive*. We can see for example that the Sex feature contributes negatively to this prediction whereas the remainder of the features have a positive contribution (i.e., increase the mortality risk). For explaining this particular data point, the Blood Pressure feature seems to have the largest effect, and corresponds to an increase in the mortality risk. See our example on how to perform explanations with this algorithm and visualise the results using the [shap library visualisations here](#) and [here](#).

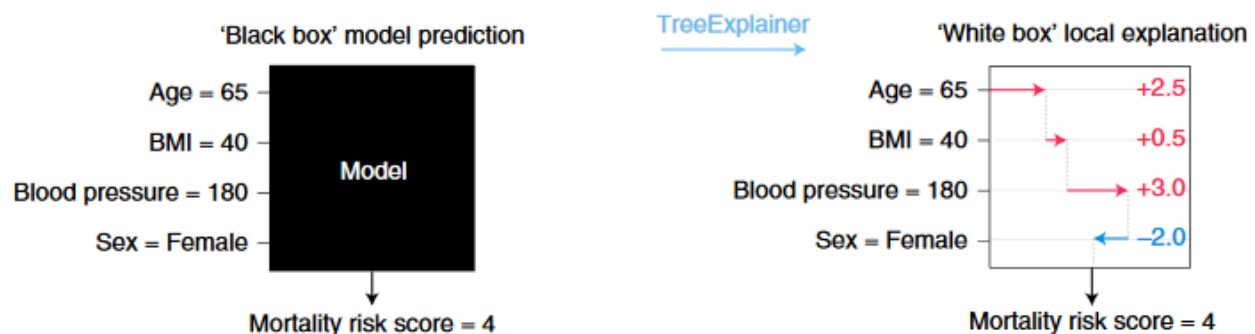


Figure 1: Cartoon illustration of explanation models with Tree SHAP.

Image Credit: Scott Lundberg (see source [here](#))

7.13.2 Usage

In order to compute the shap values , the following arguments can optionally be set when calling the explain method:

- **interactions**: set to True to decompose the shap value of every feature for every example into a main effect and interaction effects
- **approximate**: set to True to calculate an approximation to shap values (see our [example](#))
- **check_additivity**: if the explainer is initialised with `model_output = raw` and this option is True the explainer checks that the sum of the shap values is equal to model output - expected value
- **tree_limit**: if an int is passed, an ensemble formed of only `tree_limit` trees is explained

If the dataset contains categorical variables that have been encoded before being passed to the explainer and a single shap value is desired for each categorical variable, the the following options should be specified:

- **summarise_result**: set to True
- **cat_vars_start_idx**: a sequence of integers containing the column indices where categorical variables start. If the feature matrix contains a categorical feature starting at index 0 and one at index 10, then `cat_vars_start_idx=[0, 10]`
- **cat_vars_enc_dim**: a list containing the dimension of the encoded categorical variables. The number of columns specified in this list is summed for each categorical variable starting with the corresponding index in `cat_vars_start_idx`. So if `cat_vars_start_idx=[0, 10]` and `cat_vars_enc_dim=[3, 5]`, then the columns with indices 0, 1 and 2 and 10, 11, 12, 13 and 14 will be combined to return one shap value for each categorical variable, as opposed to 3 and 5.

Path-dependent feature perturbation algorithm

Initialiastion and fit

The explainer is initialised with the following arguments:

- a model, which could be an `sklearn`, `xgboost`, `catboost` or `lightgbm` model. Note that some of the models in these packages or models trained with specific objectives may not be supported. In particular, passing raw strings as categorical levels for `catboost` and `lightgbm` is not supported
- **model_output** should always default to `raw` for this algorithm
- optionally, set **task** to `'classification'` or `'regression'` to indicate the type of prediction the model makes. If set to `regression` the `prediction` field of the response is empty
- optionally, a list of feature names via **feature_names**. This is used to provide information about feature importances in the response
- optionally, a dictionary, **category_names**, that maps the columns of the categorical variables to a list of strings representing the names of the categories. This may be used for visualisation in the future.

```
from alibi.explainers import TreeShap

explainer = TreeShap(
    model,
    feature_names=['size', 'age'],
    categorical_names={0: ['S', 'M', 'L', 'XL', 'XXL']}
)
```

For this algorithm, fit is called with no arguments:

```
explainer.fit()
```

Explanation

To explain an instance X, we simply pass it to the explain method:

```
explanation = explainer.explain(X)
```

The returned explanation object has the following fields:

- `explanation.meta`:

```
{'name': 'TreeShap',  
 'type': ['whitebox'],  
 'task': 'classification',  
 'explanations': ['local', 'global'],  
 'params': {'summarise_background': False, 'algorithm': 'tree_path_dependent', 'kwargs':  
 → {}}  
}
```

This field contains metadata such as the explainer name and type as well as the type of explanations this method can generate. In this case, the `params` attribute shows the Tree SHAP variant that will be used to explain the model in the `algorithm` attribute.

- `explanation.data`:

```
data={'shap_values': [  
    array([[ 5.0661433e-01,  2.7620478e-02],  
          [-4.1725192e+00,  4.4859368e-03],  
          [ 4.1338313e-01, -5.5618007e-02]],  
    dtype=float32)  
],  
 'shap_interaction_values': [array([], dtype=float64)],  
 'expected_value': array([-0.06472124]),  
 'model_output': 'raw',  
 'categorical_names': {0: ['S', 'M', 'L', 'XL', 'XXL']},  
 'feature_names': ['size', 'age'],  
 'raw': {  
     'raw_prediction': array([-0.73818872, -8.8434663, -3.24204564]),  
     'loss': [],  
     'prediction': array([0, 0, 0]),  
     'instances': array([[0, 23],  
 [4, 55],  
 [2, 43]]),  
     'labels': array([], dtype=float64),  
     'importances': {  
         '0': {  
             'ranked_effect': array([1.6975055, 1.3598266], dtype=float32),  
             'names': [  
                 'size',  
                 'age',  
             ]  
         }  
     }  
 },  
 }
```

(continues on next page)

(continued from previous page)

```

        'aggregated': {
            'ranked_effect': array([1.6975055 , 1.3598266], dtype=float32),
            'names': [
                'size',
                'age',
            ]
        }
    }
}

```

This field contains:

- **shap_values**: a list of length equal to the number of model outputs, where each entry is an array of dimension samples x features of shap values. For the example above , 3 instances with 2 features has been explained so the shap values for each class are of dimension 3 x 2
- **shap_interaction_values**: an empty list since this `interactions` was set to `False` in the `explain` call
- **expected_value**: an array containing expected value for each model output
- **model_output**: `raw` indicates that the model raw output was explained, the only option for the path dependent algorithm
- **feature_names**: a list with the feature names
- **categorical_names**: a mapping of the categorical variables (represented by indices in the `shap_values` columns) to the description of the category
- **raw**: this field contains:
 - **raw_prediction**: a samples x n_outputs array of predictions for each instance to be explained.
 - **prediction**: an array containing the index of the maximum value in the `raw_prediction` array
 - **instances**: a samples x n_features array of instances which have been explained
 - **labels**: an array containing the labels for the instances to be explained
 - **importances**: a dictionary where each entry is a dictionary containing the sorted average magnitude of the shap value (`ranked_effect`) along with a list of feature names corresponding to the re-ordered shap values (`names`). There are `n_outputs + 1` keys, corresponding to `n_outputs` and the aggregated output (obtained by summing all the arrays in `shap_values`)

Please see our examples on how to visualise these outputs using the `shap` library visualisations [library visualisations here](#) and [here](#).

Shapley interaction values

Initialisation and fit

Shapley interaction values can only be calculated using the path-dependent feature perturbation algorithm in this release, so no arguments are passed to the `fit` method:

```

explainer = TreeShap(
    model,
    model_output='raw',

```

(continues on next page)

(continued from previous page)

```
)  
  
explainer.fit()
```

Explanation

To obtain the Shapley interaction values, the `explain` method is called with the option `interactions=True`:

```
explanation = explainer.explain(X, interactions=True)
```

The `explanation` contains a list with the shap interaction values for each model output in the `shap_interaction_values` field of the data property.

Interventional feature perturbation algorithm

Explaining model output

Initialisation and fit

```
explainer = TreeShap(  
    model,  
    model_output='raw',  
)  
  
explainer.fit(X_reference)
```

Model output can be set to `model_output='probability'` to explain models which return probabilities. Note that this requires the model to be trained with specific objectives. Please see the footnote to our path-dependent feature perturbation [example](#) for an example of how to set the model training objective in order to explain probability outputs.

Explanation

To explain instances in `X`, the explainer is called as follows:

```
explanation = explainer.explain(X)
```

Explaining loss functions

Initialisation and fit

To explain loss function, the following configuration and fit steps are necessary:

```
explainer = TreeShap(  
    model,  
    model_output='log_loss',  
)
```

(continues on next page)

(continued from previous page)

```
explainer.fit(X_reference)
```

Only square loss regression objectives and cross-entropy classification objectives are supported in this release.

Explanation

Note that the labels need to be passed to the `explain` method in order to obtain the explanation:

```
explanation = explainer.explain(X, y)
```

Miscellaneous

Runtime considerations

Adjusting the size of the reference dataset

The algorithm automatically warns the user if a background dataset size of more than 1000 samples is passed. If the runtime of an explanation with the original dataset is too large, then the algorithm can automatically subsample the background dataset during the `fit` step. This can be achieved by specifying the `fit` step as

```
explainer.fit(  
    X_reference,  
    summarise_background=True,  
    n_background_samples=300,  
)
```

or

```
explainer.fit(  
    X_reference,  
    summarise_background='auto'  
)
```

The `auto` option will select 1000 examples, whereas using the boolean argument allows the user to directly control the size of the reference set. If categorical variables are specified, the algorithm uses subsampling of the data. Otherwise, a kmeans clustering algorithm is used to select the background dataset.

As describe above, the explanations are performed with respect to the expected output over this dataset so the shap values will be affected by the dataset selection. We recommend experimenting with various ways to choose the background dataset before deploying explanations.

7.13.3 Theoretical overview

Recall that, for a model f , the Kernel SHAP algorithm [1] explains a certain outcome with respect to a chosen reference (or an expected value) by estimating the shap values of each feature i from $\{1, \dots, M\}$, as follows:

- enumerate all subsets S of the set $F \setminus \{i\}$
- for each $S \subseteq F \setminus \{i\}$, compute the contribution of feature i as $C(i|S) = f(S \cup \{i\}) - f(S)$
- compute the shap value according to

$$\phi_i := \frac{1}{M} \sum_{S \subseteq F \setminus \{i\}} \frac{1}{\binom{M-1}{|S|}} C(i|S). \quad (1)$$

Since most models do not accept arbitrary patterns of missing values at inference time, $f(S)$ needs to be approximated. The original formulation of the Kernel Shap algorithm [1] proposes to compute $f(S)$ as the *observational conditional expectation*

$$f(S) := \mathbb{E}[f(\mathbf{x}_S, \mathbf{X}_{\bar{S}} | \mathbf{X}_S = \mathbf{x}_S)] \quad (2)$$

where the expectation is taken over a *background dataset*, \mathcal{D} , after conditioning. Computing this expectation involves drawing sufficiently many samples from $\mathbf{X}_{\bar{S}}$ for every sample from \mathbf{X}_S , which is expensive. Instead, (2) is approximated by

$$f(S) := \mathbb{E}[f(\mathbf{x}_S, \mathbf{X}_{\bar{S}})]$$

where features in a subset S are fixed and features in \bar{S} are sampled from the background dataset. This quantity is referred to as *marginal* or *interventional conditional expectation*, to emphasise that setting features in S to the values \mathbf{x}_S can be viewed as an intervention on the instance to be explained.

As described in [2], if estimating impact of a feature i on the function value by $\mathbb{E}[f | X_i = x_i]$, one should bear in mind that observing $X_i = x_i$ changes the distribution of the features $X_{j \neq i}$ if these variables are correlated. Hence, if the conditional expectation is used to estimate $f(S)$, the Shapley values might not be accurate since they also depend on the remaining variables, effect which becomes important if there are strong correlations amongst the independent variables. Furthermore, the authors show that estimating $f(S)$ using the conditional expectation violates the *sensitivity principle*, according to which the Shapley value of a redundant variable should be 0. On the other hand, the intervention breaks the dependencies, ensuring that the sensitivity holds. One potential drawback of this method is that setting a subset of values to certain values without regard to the values of the features in the complement (i.e., \bar{S}) can generate instances that are outside the training data distribution, which will affect the model prediction and hence the contributions.

The following sections detail how these methods work and how, unlike Kernel SHAP, compute the exact shap values in polynomial time. The algorithm estimating contributions using interventional expectations is presented, with the remaining sections being dedicated to presenting an approximate algorithm for evaluating the interventional expectation that does not require a background dataset and Shapley interaction values.

Interventional feature perturbation

The interventional feature perturbation algorithm provides an efficient way to calculate the expectation $f(S) := \mathbb{E}[f(\mathbf{x}_S, \mathbf{X}_{\bar{S}})]$ for all possible subsets S , and to combine these values according to equation (1) in order to obtain the Shapley value. Intuitively, one can proceed as follows:

- choose a background sample $r \in \mathcal{D}$
- for each feature i , enumerate all subsets $S \subseteq F \setminus \{i\}$
- for each such subset, S , compute $f(S)$ by traversing the tree with a *hybrid sample* where the features in \bar{S} are replaced by their corresponding values in r

- combine results according to equation (1)

If R samples from the background distribution are used, then the complexity of this algorithm is $O(RM2^M)$ since we perform 2^M enumerations for each of the M features, R times. The key insight into this algorithm is that multiple hybrid samples will end up traversing identical paths and that this can be avoided if the shap values' calculation is reformulated as a summation over the paths in the tree (see [4] for a proof):

$$\phi_i = \sum_P \phi_i^P$$

where the summation is over paths P in the tree descending from i . The value and sign of the contribution of each path descending through a node depends on whether the split from the node is due to a foreground or a background feature, as explained in the practical example below.

Computing contributions with interventional Tree SHAP: a practical example.

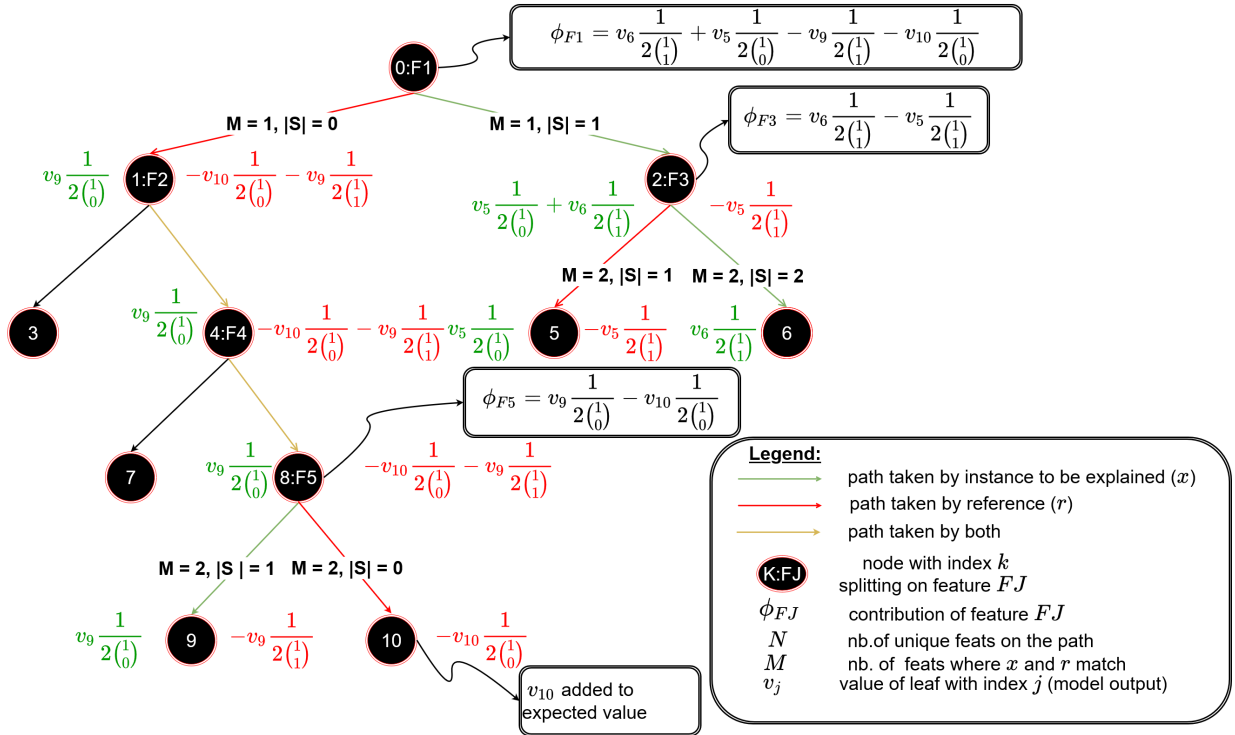


Figure 2: Illustration of the feature contribution and expected value estimation process using interventional perturbation Tree SHAP. The positive and the negative contributions of a node are represented in green and red, respectively.

In the figure above, the paths followed due the instance to be explained x are coloured in red, paths followed due to the background sample in red, and common paths in yellow.

The instance to be explained is perturbed using a reference sample by the values of the features $F1$, $F3$ and $F5$ in x with the corresponding values in r . This process gives the name of the algorithm since following the paths indicated by the background sample is akin to intervening on the instance to be explained with features from the background sample. Therefore, one defines the set F in the previous section as $F = \{j : x_j \neq r_j\}$ for this case. Note that these are the only features for which one can estimate a contribution given this background sample; the same path is followed for features $F2$ and $F4$ for both the original and the perturbed sample, so these features do not contribute to explain the difference between the observed outcome (v_6) and the outcome that would have been observed if the tree had been traversed according to the reference (v_{10}).

Considering the structure of the tree for the given x and r together with equation (1) reveals that the left subtree can be traversed to compute the negative terms in the summation whereas the right subtree will provide positive terms. This is because the nodes in the left subtree can only be reached if $F1$ takes the value from the background sample, that is, only $F1$ is missing. Because $F2$ and $F4$ do not contribute to explaining $f(x) - f(r)$, the negative contribution of the left subtree will be equal to the negative contribution of node 8. This node sums two negative components: one when the downstream feature $F5$ is also missing (corresponding to evaluating f at $S = \emptyset$) and one when $F5$ is present (corresponding to evaluating f at $S = \{F5\}$). These negative values are weighted according to the combinatorial factor in equation (1). By a similar reasoning, the nodes in the right subtree are reached only if $F1$ is present and they provide the positive terms for the shap value computation. Note that the combinatorial factor in (1) should be evaluated with $|S| \leftarrow |S| - 1$ for positive contributions since $|S|$ is increased by 1 because of the feature whose contribution is calculated is present in the right subtree.

A similar reasoning is applied to compute the contributions of the downstream nodes. For example, to estimate the contribution of $F5$, one considers a set $S = \emptyset$ and observes the value of node 10, and weighs that with the combinatorial factor from equation (1) where $M - 1 = 1$ and $|S| = 0$ (because there are no features present on the path) and a positive contribution from node 9 weighted by the same combinatorial factor (because $S = \{F5\}$ so $|S| - 1 = 0$).

To summarise, the efficient algorithm relies on the following key ideas:

- each node in the tree is assigned a positive contribution reflecting membership of the splitting feature in a subset S and a negative contribution to indicate the feature is missing ($i \in \bar{S}$)
- the positive and negative contributions of a node can be computed by summing the positive and negative contributions of the children nodes, in keeping with the fact that the Shapley value can be computed by summing a contribution from each path the feature is on
- to compute the contribution of a feature at a node, one adds a positive contribution from the node reached by splitting on the feature from the instance to be explained and a negative contribution from the node reached by splitting on the feature in the background sample
- features for which the instance to be explained and the reference follow the same path are assigned 0 contribution.

Explaining loss functions

One advantage of the interventional approach is that it allows to approximately transform the shap values to account for nonlinear transformation of outputs, such as the loss function. Recall that given ϕ_i, \dots, ϕ_M the local accuracy property guarantees that given $\phi_0 = \mathbb{E}[f(x)]$

$$f(x) = \phi_0 + \sum_{i=1}^M \phi_i. \quad (3)$$

Hence, in order to account for the effect of the nonlinear transformation h , one has to find the functions g_0, \dots, g_M such that

$$h(f(x)) = g(\phi_0) + \sum_{i=1}^M g_i(\phi_i) \quad (4)$$

For simplicity, let $y = h(x)$. Then using a first-order Taylor series expansion around $\mathbb{E}[y]$ one obtains

$$h(y) \approx h(\mathbb{E}[y]) + \left. \frac{\partial h(y)}{\partial y} \right|_{y=\mathbb{E}[y]} (y - \mathbb{E}[y]). \quad (5)$$

Substituting (3) in (5) and comparing coefficients with (4) yields

$$g_0 \approx h(\mathbb{E}[y])$$

$$g_i \approx \phi_i \frac{\partial h(y)}{\partial y} \Big|_{y=\mathbb{E}[y]}.$$

Hence, an approximate correction is given by simply scaling the shap values using the gradient of the nonlinear function. Note that in practice one may take the Taylor series expansion at a reference point r from the background dataset and average over the entire background dataset to compute the scaling factor. This introduces an additional source of noise since $h(\mathbb{E}[y]) = \mathbb{E}[h(y)]$ only when h is linear.

Computational complexity

For a single foreground and background sample and a single tree, the algorithm runs in $O(LD)$ time. Thus, using R background samples and a model containing T trees, yields a complexity of $O(TRL D)$.

Path dependent feature perturbation

Another way to approximate equation (2) to compute $f(S)$ given an instance x and a set of missing features \bar{S} is to recursively follow the decision path through the tree and:

- return the node value if a split on a feature $i \in S$ is performed
- take a weighted average of the values returned by children if $i \in \bar{S}$, where the weighing factor is equal to the proportion of training examples flowing down each branch. This proportion is a property of each node, sometimes referred to as *weight* or *cover* and measures how important is that node with regard to classifying the training data.

Therefore, in the path-dependent perturbation method, we compute the expectations with respect to the training data distribution by weighting the leaf values according to the proportion of the training examples that flow to that leaf.

To avoid repeating the above recursion $M2^M$ times, one first notices that for a single decision tree, applying a perturbation would result in the sample ending up in a different leaf. Therefore, following each path from the root to a leaf in the tree is equivalent to perturbing subsets of features of varying cardinalities. Consequently, each leaf will contain a certain proportion of all possible subsets $S \subseteq F$. Therefore, to compute the shap values, the following quantities are computed at each leaf, *for every feature i on the path leading to that leaf*:

- the proportion of subsets S at the leaf that contain i and the proportion of subsets S that do not contain i
- for each cardinality, the proportion of the sets of that cardinality contained at the leaf. Tracking each cardinality as opposed to a single count of subsets falling into a given leaf is necessary since it allows to apply the weighting factor in equation (1), which depends on the subset size, $|S|$.

This intuition can be summarised as follows:

$$\phi_i := \sum_{j=1}^L \sum_{P \in S_j} \frac{w(|P|, j)}{M_j \binom{M_j-1}{|P|}} (p_o^{i,j} - p_z^{i,j}) v_j \quad (6)$$

where S_j is the set of present feature subsets at leaf j , M_j is the length of the path and $w(|P|, j)$ is the proportion of all subsets of cardinality P at leaf j , $p_o^{i,j}$ and $p_z^{i,j}$ represent the fractions of subsets that contain or do not contain feature i respectively.

Computational complexity

Using the above quantities, one can compute the *contribution* of each leaf to the Shapley value of every feature. This algorithm has complexity $O(TLD^2)$ for an ensemble of trees where L is the number of leaves, T the number of trees in the ensemble and D the maximum tree depth. If the tree is balanced, then $D = \log L$ and the complexity of our algorithm is $O(TL \log^2 L)$.

Expected value for the path-dependent perturbation algorithm

Note that although a background dataset is not provided, the expected value is computed using the node cover information, stored at each node. The computation proceeds recursively, starting at the root. The contribution of a node to the expected value of the tree is a function of the expected values of the children and is computed as follows:

$$c_j = \frac{c_{r(j)}r_{r(j)} + c_{l(j)}r_{l(j)}}{r_j}$$

where j denotes the node index, c_j denotes the node expected value, r_j is the cover of the j th node and $r(j)$ and $l(j)$ represent the indices of the right and left children, respectively. The expected value used by the tree is simply c_{root} . Note that for tree ensembles, the expected values of the ensemble members is weighted according to the tree weight and the weighted expected values of all trees are summed to obtain a single value.

The cover depends on the objective function and the model chosen. For example, in a gradient boosted tree trained with squared loss objective, r_j is simply the number of training examples flowing through j . For an arbitrary objective, this is the sum of the Hessian of the loss function evaluated at each point flowing through j , as explained here.

Shapley interaction values

While the Shapley values provide a solution to the problem of allocating a function variation to the input features, in practice it might be of interest to understand how the importance of a feature depends on the other features. The Shapley interaction values can solve this problem, by allocating the change in the function amongst the individual features (*main effects*) and all pairs of features (*interaction effects*). Thus, they are defined as

$$\Phi_{i,j}(f, x) = \sum_{S \subseteq F \setminus \{i,j\}} \frac{1}{2|S| \binom{M-1}{|S|-1}} \nabla_{ij}(f, x, S), \quad i \neq j \quad (7)$$

and

$$\nabla_{ij}(f, x, S) = \underbrace{f_x(S \cup \{i, j\}) - f_x(S \cup \{j\})}_{j \text{ present}} - \underbrace{[f_x(S \cup \{i\}) - f_x(S)]}_{j \text{ not present}}. \quad (8)$$

Therefore, the interaction of features i and j can be computed by taking the difference between the shap values of i when j is present and when j is not present. The main effects are defined as

$$\Phi_{i,i}(f, x) = \phi_i(f, x) - \sum_{i \neq j} \Phi_{i,j}(f, x),$$

Setting $\Phi_{0,0} = f_x(\emptyset)$ yields the local accuracy property for Shapley interaction values:

$$f(x) = \sum_{i=0}^M \sum_{j=0}^M \Phi_{i,j}(f, x)$$

.

The interaction is split equally between feature i and j , which is why the division by two appears in equation (7). The total interaction effect is defined as $\Phi_{i,j}(f, x) + \Phi_{j,i}(f, x)$.

Computational complexity

According to equation (8), the interaction values can be computed by applying either the interventional or path-dependent feature perturbation algorithm twice: once by fixing the value of feature j to x_j and computing the shapley value for feature i in this configuration, and once by fixing x_j to a “missing” value and performing the same computation. Thus, the interaction values can be computed in $O(TMLD^2)$ with the path-dependent perturbation algorithm and $O(TMLDR)$ with the interventional feature perturbation algorithm.

Comparison to other methods

Tree-based models are widely used in areas where model interpretability is of interest because node-level statistics gathered from the training data can be used to provide insights into the behaviour of the model across the training dataset, providing a *global explanation* technique. As shown in our [example](#), considering different statistics gives rise to different importance rankings. As discussed in [\[1\]](#) and [\[3\]](#), depending on the statistic chosen, feature importances derived from trees are not *consistent*, meaning that a model where a feature is known to have a bigger impact might fail to have a larger importance. As such, feature importances cannot be compared across models. In contrast, both the path-dependent and interventional perturbation algorithms tackle this limitation.

In contrast to feature importances derived from tree statistics, the Tree SHAP algorithms can also provide local explanations, allowing the identification of features that are globally “not important”, but can affect specific outcomes significantly, as might be the case in healthcare applications. Additionally, it provides a means to succinctly summarise the effect magnitude and direction (positive or negative) across potentially large samples. Finally, as shown in [\[1\]](#) (see [here](#), p. 26), averaging the instance-level shap values importance to derive a global score for each feature can result in improvements in feature selection tasks.

Another method to derive instance-level explanations for tree-based model has been proposed by Sabaas [here](#). This feature attribution method is similar in spirit to Shapley value, but does not account for the effect of variable order as explained [here](#) (pp. 10-11) as well as not satisfying consistency ([\[3\]](#)).

Finally, both Tree SHAP algorithms exploit model structure to provide exact Shapley values computation albeit using different estimates for the effect of missing features, achieving explanations in low-order polynomial time. The KernelShap method relies on post-hoc (black-box) function modelling and approximations to approximate the same quantities and given enough samples has been shown to to the exact values (see experiments [here](#) and our [example](#)). Our Kernel SHAP [documentation](#) provides comparisons of feature attribution methods based on Shapley values with other algorithms such as LIME and [anchors](#).

7.13.4 References

- [1] Lundberg, S.M. and Lee, S.I., 2017. A unified approach to interpreting model predictions. In Advances in neural information processing systems (pp. 4765-4774).
- [2] Janzing, D., Minorics, L. and Blöbaum, P., 2019. Feature relevance quantification in explainable AI: A causality problem. arXiv preprint arXiv:1910.13413.
- [3] Lundberg, S.M., Erion, G.G. and Lee, S.I., 2018. Consistent individualized feature attribution for tree ensembles. arXiv preprint arXiv:1802.03888.
- [4] Chen, H., Lundberg, S.M. and Lee, S.I., 2018. Understanding Shapley value explanation algorithms for trees. Under review for publication in Distill, draft available [here](#).

7.13.5 Examples

Path-dependent Feature Perturbation Tree SHAP

Explaining tree models with path-dependent feature perturbation Tree SHAP

Interventional Feature Perturbation Tree SHAP

Explaining tree models with path-dependent feature perturbation Tree SHAP

EXAMPLES

8.1 Alibi Overview Example

This notebook aims to demonstrate each of the explainers Alibi provides on the same model and dataset. Unfortunately, this isn't possible as white-box neural network methods exclude tree-based white-box methods. Hence we will train both a neural network(TensorFlow) and a random forest model on the same dataset and apply the full range of explainers to see what insights we can obtain.

The results and code from this notebook are used in the [documentation overview](#).

This notebook requires the seaborn package for visualization which can be installed via pip:

Note

To ensure all dependencies are met for this example, you may need to run

```
pip install alibi[all]
```

```
[2]: !pip install -q seaborn
```

```
[3]: import requests
from io import BytesIO, StringIO
from io import BytesIO
from zipfile import ZipFile

import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
import seaborn as sns
sns.set(rc={'figure.figsize':(11.7,8.27)})

import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import joblib
import os.path

import tensorflow as tf
np.random.seed(0)
```

(continues on next page)

(continued from previous page)

```
tf.random.set_seed(0)

FROM_SCRATCH = False
TF_MODEL_FNAME = 'tf-clf-wine'
RFC_FNAME = 'rfc-wine'
ENC_FNAME = 'wine_encoder'
DEC_FNAME = 'wine_decoder'
```

8.1.1 Preparing the data.

We're using the [wine-quality](#) dataset, a numeric tabular dataset containing features that refer to the chemical composition of wines and quality ratings. To make this a simple classification task, we bucket all wines with ratings greater than five as good, and the rest we label bad. We also normalize all the features.

```
[4]: def fetch_wine_ds():
      url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/
      ↪winequality-red.csv'
      resp = requests.get(url, timeout=2)
      resp.raise_for_status()
      string_io = StringIO(resp.content.decode('utf-8'))
      return pd.read_csv(string_io, sep=';')
```

```
[5]: df = fetch_wine_ds()
```

```
[6]: df['class'] = 'bad'
      df.loc[(df['quality'] > 5), 'class'] = 'good'

      features = [
          'fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
          'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
          'pH', 'sulphates', 'alcohol'
      ]

      df['good'] = 0
      df['bad'] = 0
      df.loc[df['class'] == 'good', 'good'] = 1
      df.loc[df['class'] == 'bad', 'bad'] = 1

      data = df[features].to_numpy()
      labels = df[['class', 'good', 'bad']].to_numpy()

      X_train, X_test, y_train, y_test = train_test_split(data, labels, random_state=0)
      X_train, X_test = X_train.astype('float32'), X_test.astype('float32')
      y_train_lab, y_test_lab = y_train[:, 0], y_test[:, 0]
      y_train, y_test = y_train[:, 1:].astype('float32'), y_test[:, 1:].astype('float32')

      scaler = StandardScaler()
      scaler.fit(X_train)
```

```
[6]: StandardScaler()
```

Select good wine instance

We partition the dataset into good and bad portions and select an instance of interest. I've chosen it to be a good quality wine.

Note that bad wines are class 1 and correspond to the second model output being high, whereas good wines are class 0 and correspond to the first model output being high.

```
[7]: bad_wines = np.array([a for a, b in zip(X_train, y_train) if b[1] == 1])
good_wines = np.array([a for a, b in zip(X_train, y_train) if b[1] == 0])
x = np.array([9.2, 0.36, 0.34, 1.6, 0.062, 5., 12., 0.99667, 3.2, 0.67, 10.5]]) #
↳ prechosen instance
```

8.1.2 Training models

Creating an Autoencoder

For some of the explainers, we need an autoencoder to check whether example instances are close to the training data distribution or not.

```
[8]: from tensorflow.keras.layers import Dense
from tensorflow import keras

ENCODING_DIM = 7
BATCH_SIZE = 64
EPOCHS = 100

class AE(keras.Model):
    def __init__(self, encoder: keras.Model, decoder: keras.Model, **kwargs) -> None:
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def call(self, x: tf.Tensor, **kwargs):
        z = self.encoder(x)
        x_hat = self.decoder(z)
        return x_hat

def make_ae():
    len_input_output = X_train.shape[-1]

    encoder = keras.Sequential()
    encoder.add(Dense(units=ENCODING_DIM*2, activation="relu", input_shape=(len_input_
↳ output, )))
    encoder.add(Dense(units=ENCODING_DIM, activation="relu"))

    decoder = keras.Sequential()
    decoder.add(Dense(units=ENCODING_DIM*2, activation="relu", input_shape=(ENCODING_DIM,
↳ )))
    decoder.add(Dense(units=len_input_output, activation="linear"))
```

(continues on next page)

(continued from previous page)

```

ae = AE(encoder=encoder, decoder=decoder)

ae.compile(optimizer='adam', loss='mean_squared_error')
history = ae.fit(
    scaler.transform(X_train),
    scaler.transform(X_train),
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    verbose=False,)

# loss = history.history['loss']
# plt.plot(loss)
# plt.xlabel('Epoch')
# plt.ylabel('MSE-Loss')

ae.encoder.save(f'{ENC_FNAME}.h5')
ae.decoder.save(f'{DEC_FNAME}.h5')
return ae

def load_ae_model():
    encoder = load_model(f'{ENC_FNAME}.h5')
    decoder = load_model(f'{DEC_FNAME}.h5')
    return AE(encoder=encoder, decoder=decoder)

```

Random Forest Model

We need a tree-based model to get results for the tree SHAP explainer. Hence we train a random forest on the wine-quality dataset.

```

[9]: from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score

def make_rfc():
    rfc = RandomForestClassifier(n_estimators=50)
    rfc.fit(scaler.transform(X_train), y_train_lab)
    y_pred = rfc.predict(scaler.transform(X_test))

    print('accuracy_score:', accuracy_score(y_pred, y_test_lab))
    print('f1_score:', f1_score(y_test_lab, y_pred, average=None))

    joblib.dump(rfc, f'{RFC_FNAME}.joblib')
    return rfc

def load_rfc_model():
    return joblib.load(f'{RFC_FNAME}.joblib')

```

Tensorflow Model

Finally, we also train a TensorFlow model.

```
[10]: from tensorflow import keras
      from tensorflow.keras import layers

      def make_tf_model():
          inputs = keras.Input(shape=X_train.shape[1])
          x = layers.Dense(6, activation="relu")(inputs)
          outputs = layers.Dense(2, activation="softmax")(x)
          model = keras.Model(inputs, outputs)

          model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=['accuracy',
↪'])
          history = model.fit(
              scaler.transform(X_train),
              y_train,
              epochs=30,
              verbose=False,
              validation_data=(scaler.transform(X_test), y_test),
          )

          y_pred = model(scaler.transform(X_test)).numpy().argmax(axis=1)
          print('accuracy_score:', accuracy_score(y_pred, y_test.argmax(axis=1)))
          print('f1_score:', f1_score(y_pred, y_test.argmax(axis=1), average=None))

          model.save(f'{TF_MODEL_FNAME}.h5')
          return model

      def load_tf_model():
          return load_model(f'{TF_MODEL_FNAME}.h5')
```

Load/Make models

We save and load the same models each time to ensure stable results. If they don't exist we create new ones. If you want to generate new models on each notebook run, then set FROM_SCRATCH=True.

```
[11]: if FROM_SCRATCH or not os.path.isfile(f'{TF_MODEL_FNAME}.h5'):
      model = make_tf_model()
      rfc = make_rfc()
      ae = make_ae()
  else:
      rfc = load_rfc_model()
      model = load_tf_model()
      ae = load_ae_model()

accuracy_score: 0.74
f1_score: [0.75471698 0.72340426]
accuracy_score: 0.815
f1_score: [0.8          0.82790698]
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be
↪built. `model.compile_metrics` will be empty until you train or evaluate the model.
```

(continues on next page)

(continued from previous page)

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be
↳built. `model.compile_metrics` will be empty until you train or evaluate the model.
```

8.1.3 Util functions

These are utility functions for exploring results. The first shows two instances of the data side by side and compares the difference. We'll use this to see how the counterfactuals differ from their original instances. The second function plots the importance of each feature. This will be useful for visualizing the attribution methods.

```
[12]: def compare_instances(x, cf):
    """
    Show the difference in values between two instances.
    """
    x = x.astype('float64')
    cf = cf.astype('float64')
    for f, v1, v2 in zip(features, x[0], cf[0]):
        print(f'{f:<25} instance: {round(v1, 3):^10} counter factual: {round(v2, 3):^10}↳
↳difference: {round(v1 - v2, 7):^5}')

def plot_importance(feats_imp, feat_names, class_idx, **kwargs):
    """
    Create a horizontal barchart of feature effects, sorted by their magnitude.
    """

    df = pd.DataFrame(data=feats_imp, columns=feat_names).sort_values(by=0, axis='columns
↳')
    feats_imp, feat_names = df.values[0], df.columns
    fig, ax = plt.subplots(figsize=(10, 5))
    y_pos = np.arange(len(feats_imp))
    ax.barh(y_pos, feats_imp)
    ax.set_yticks(y_pos)
    ax.set_yticklabels(feat_names, fontsize=15)
    ax.invert_yaxis()
    ax.set_xlabel(f'Feature effects for class {class_idx}', fontsize=15)
    return ax, fig
```

8.1.4 Local Feature Attribution

Integrated Gradients

The integrated gradients (IG) method computes the attribution of each feature by integrating the model partial derivatives along a path from a baseline point to the instance. This accumulates the changes in the prediction that occur due to the changing feature values. These accumulated values represent how each feature contributes to the prediction for the instance of interest.

We illustrate the application of IG to the instance of interest.

```
[13]: from alibi.explainers import IntegratedGradients

ig = IntegratedGradients(model,
```

(continues on next page)

(continued from previous page)

```

        layer=None,
        method="gausslegendre",
        n_steps=50,
        internal_batch_size=100)

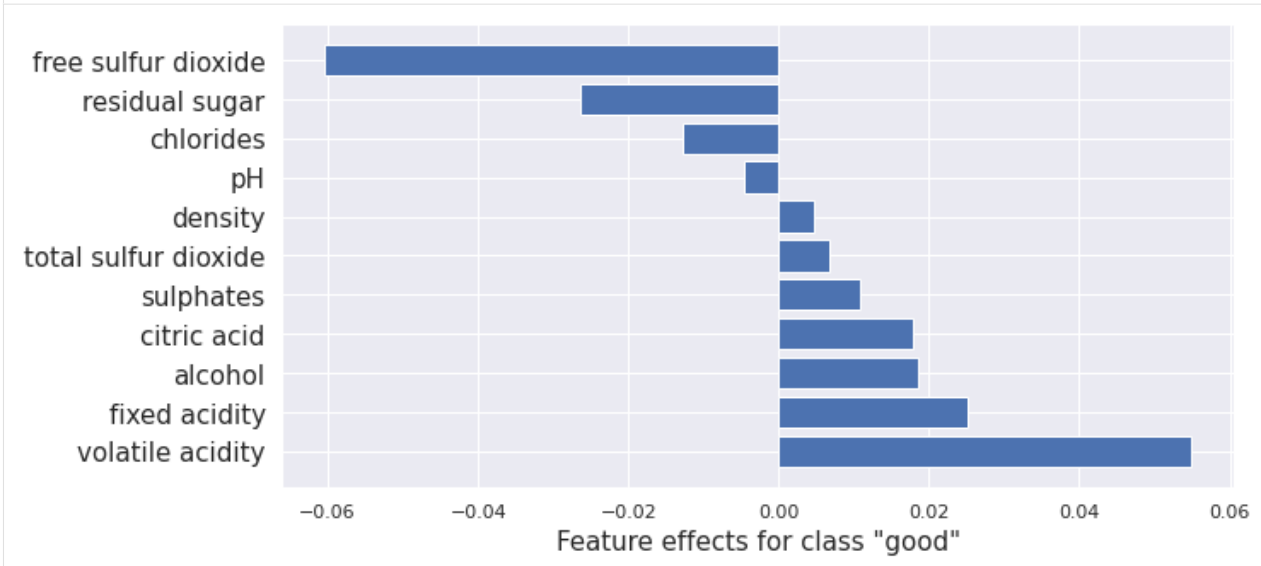
result = ig.explain(scaler.transform(x), target=0)

plot_importance(result.data['attributions'][0], features, "good")

/home/alex/.local/lib/python3.8/site-packages/tqdm/auto.py:22: TqdmWarning: IProgress
→not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/
→en/stable/user_install.html
    from .autonotebook import tqdm as notebook_tqdm

```

[13]: (<AxesSubplot:xlabel='Feature effects for class "good">,
<Figure size 720x360 with 1 Axes>)



Kernel SHAP

Kernel SHAP is a method for computing the Shapley values of a model around an instance. Shapley values are a game-theoretic method of assigning payout to players depending on their contribution to an overall goal. In our case, the features are the players, and the payouts are the attributions.

Here we give an example of Kernel SHAP method applied to the Tensorflow model.

```

[14]: from alibi.explainers import KernelShap

predict_fn = lambda x: model(scaler.transform(x))

explainer = KernelShap(predict_fn, task='classification')

explainer.fit(X_train[0:100])

result = explainer.explain(x)

```

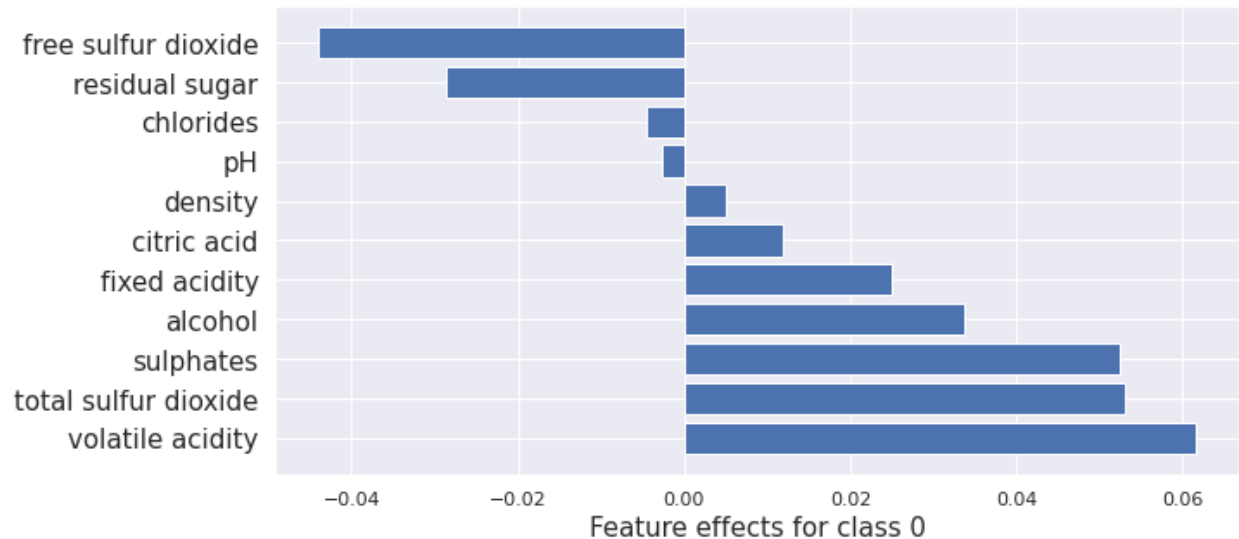
(continues on next page)

(continued from previous page)

```
plot_importance(result.shap_values[0], features, 0)
```

```
100%|=====| 1/1 [00:00<00:00, 2.55it/s]
```

```
[14]: (<AxesSubplot:xlabel='Feature effects for class 0'>,
      <Figure size 720x360 with 1 Axes>)
```



Here we apply Kernel SHAP to the Tree-based model to compare to the tree-based methods we run later.

```
[15]: from alibi.explainers import KernelShap

predict_fn = lambda x: rfc.predict_proba(scaler.transform(x))

explainer = KernelShap(predict_fn, task='classification')

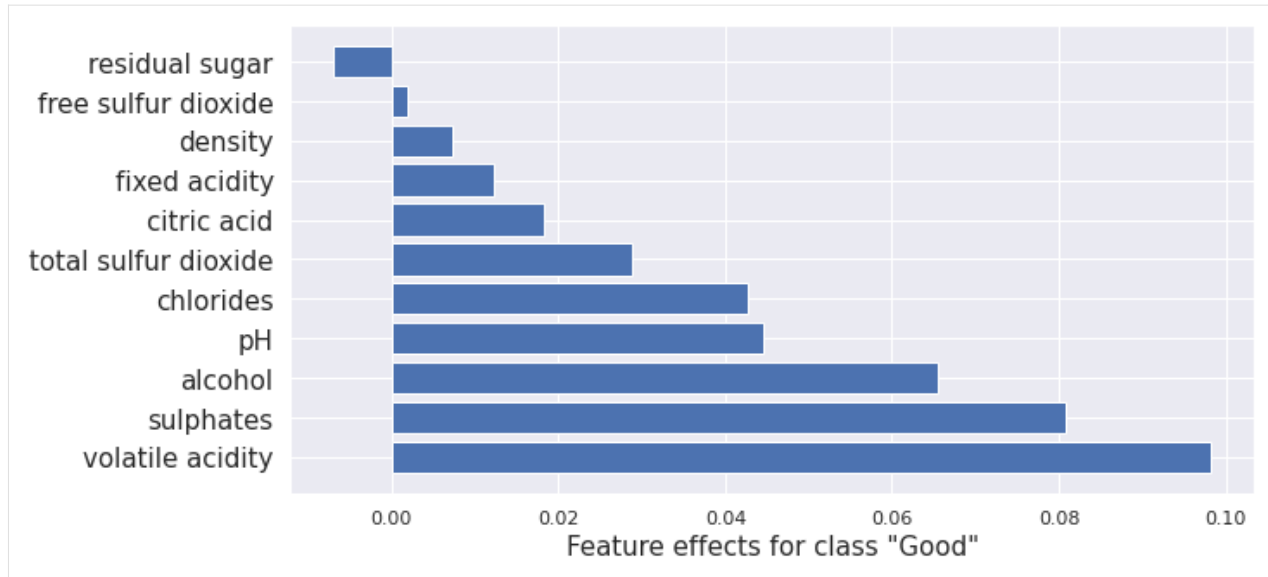
explainer.fit(X_train[0:100])

result = explainer.explain(x)

plot_importance(result.shap_values[1], features, "Good")
```

```
100%|=====| 1/1 [00:00<00:00, 1.21it/s]
```

```
[15]: (<AxesSubplot:xlabel='Feature effects for class "Good">,
      <Figure size 720x360 with 1 Axes>)
```

Interventional treeSHAP

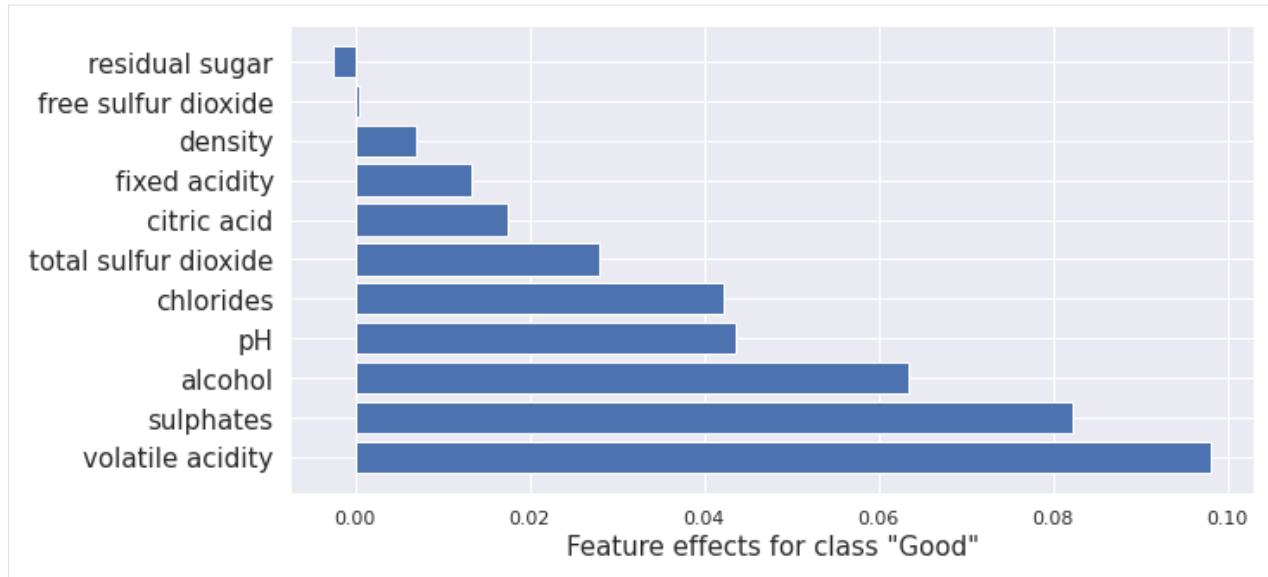
Interventional tree SHAP computes the same Shapley values as the kernel SHAP method above. The difference is that it's much faster for tree-based models. Here it is applied to the random forest we trained. Comparison with the kernel SHAP results above show very similar outcomes.

```
[16]: from alibi.explainers import TreeShap

tree_explainer_interventional = TreeShap(rfc, model_output='raw', task='classification')
tree_explainer_interventional.fit(scaler.transform(X_train[0:100]))
result = tree_explainer_interventional.explain(scaler.transform(x), check_
↪additivity=False)

plot_importance(result.shap_values[1], features, '"Good"')

[16]: (<AxesSubplot:xlabel='Feature effects for class "Good">',
      <Figure size 720x360 with 1 Axes>)
```



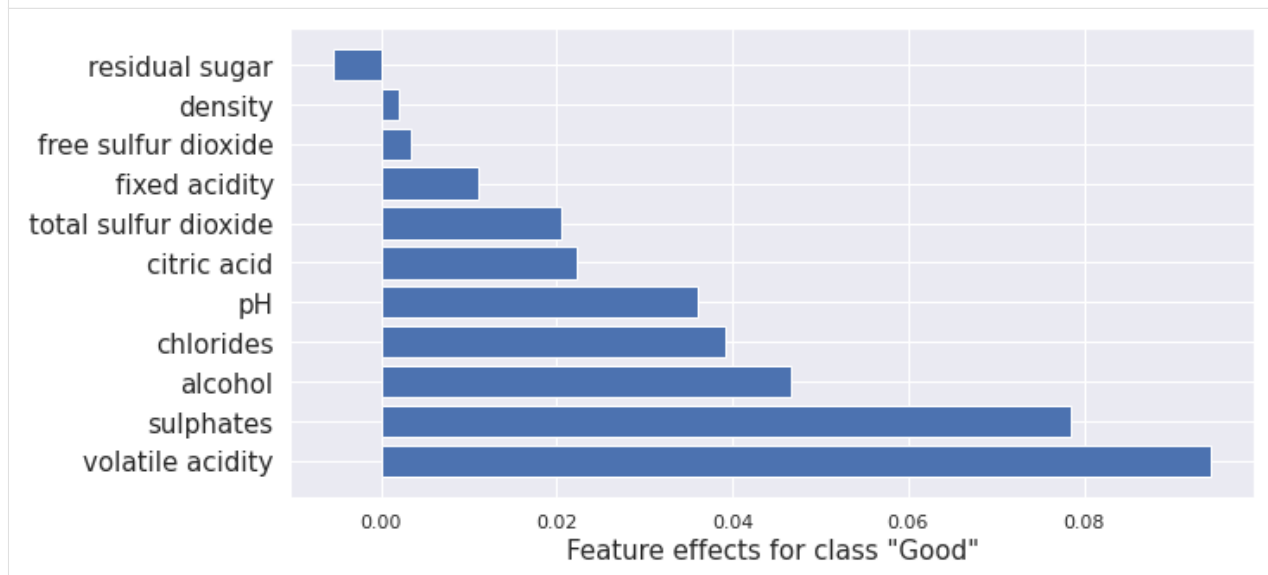
Path Dependent treeSHAP

Path Dependent tree SHAP gives the same results as the Kernel SHAP method, only faster. Here it is applied to a random forest model. Again very similar results to kernel SHAP and Interventional tree SHAP as expected.

```
[17]: path_dependent_explainer = TreeShap(rfc, model_output='raw', task='classification')
path_dependent_explainer.fit()
result = path_dependent_explainer.explain(scaler.transform(x))

plot_importance(result.shap_values[1], features, "Good")
```

```
[17]: (<AxesSubplot:xlabel='Feature effects for class "Good">,
<Figure size 720x360 with 1 Axes>)
```



Note: There is some difference between the kernel SHAP and integrated gradient applied to the TensorFlow model and the SHAP methods applied to the random forest. This is expected due to the combination of different methods and

models. They are reasonably similar overall. Notably, the ordering is nearly the same.

8.1.5 Local Necessary Features

Anchors

Anchors tell us what features need to stay the same for a specific instance for the model to give the same classification. In the case of a trained image classification model, an anchor for a given instance would be a minimal subset of the image that the model uses to make its decision.

Here we apply Anchors to the tensor flow model trained on the wine-quality dataset.

```
[18]: from alibi.explainers import AnchorTabular

predict_fn = lambda x: model.predict(scaler.transform(x))
explainer = AnchorTabular(predict_fn, features)
explainer.fit(X_train, disc_perc=(25, 50, 75))
result = explainer.explain(x, threshold=0.95)
```

The result is a set of predicates that tell you whether a point in the data set is in the anchor or not. If it is in the anchor, it is very likely to have the same classification as the instance x .

```
[19]: print('Anchor = ', result.data['anchor'])
print('Precision = ', result.data['precision'])
print('Coverage = ', result.data['coverage'])

Anchor = ['alcohol > 10.10', 'volatile acidity <= 0.39']
Precision = 0.9513641755634639
Coverage = 0.16263552960800667
```

8.1.6 Global Feature Attribution

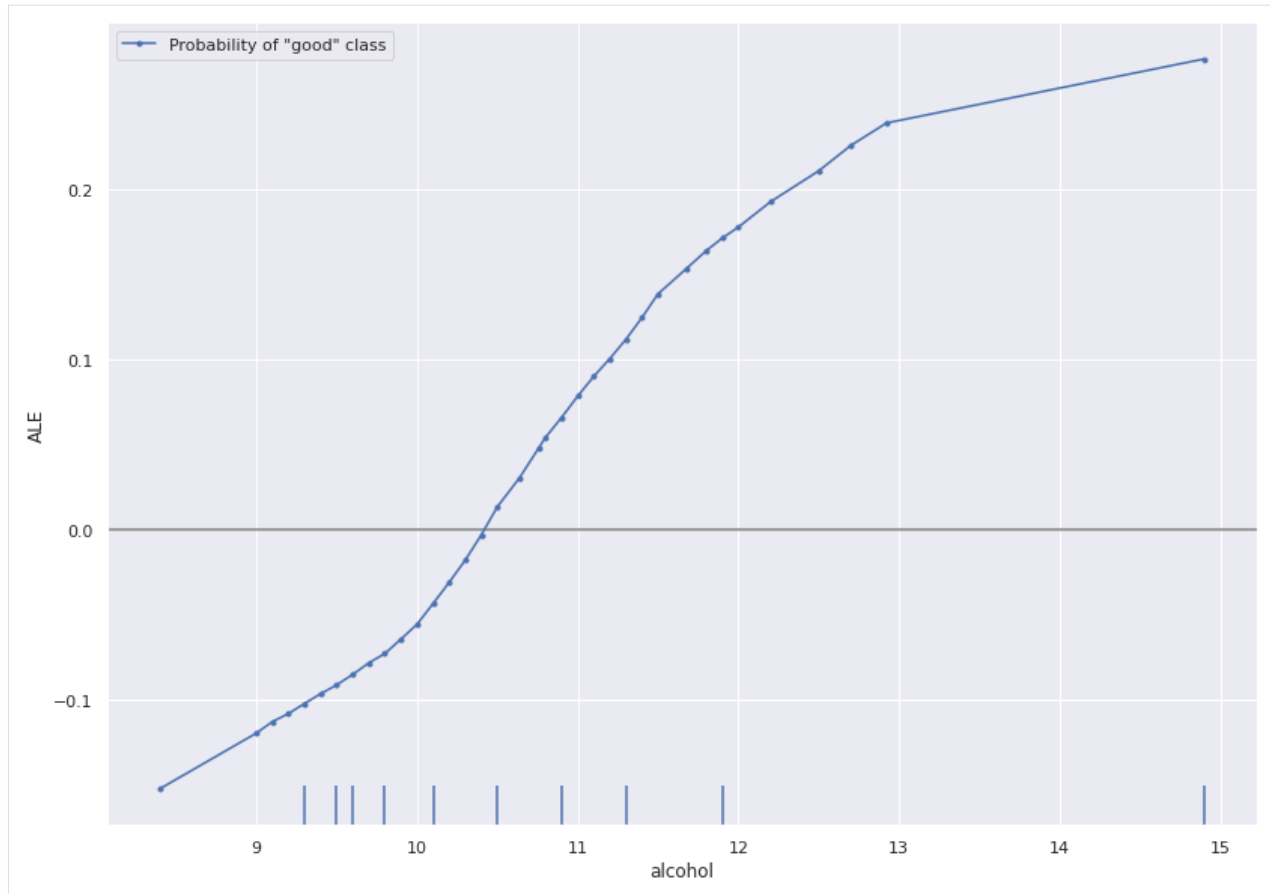
ALE

ALE plots show the dependency of model output on a subset of the input features. They provide global insight describing the model's behaviour over the input space. Here we use ALE to directly visualize the relationship between the TensorFlow model's predictions and the alcohol content of wine.

```
[20]: from alibi.explainers import ALE
from alibi.explainers.ale import plot_ale

predict_fn = lambda x: model(scaler.transform(x)).numpy()[0]
ale = ALE(predict_fn, feature_names=features)
exp = ale.explain(X_train)
plot_ale(exp, features=['alcohol'], line_kw={'label': 'Probability of "good" class'})

[20]: array([[<AxesSubplot:xlabel='alcohol', ylabel='ALE'>]], dtype=object)
```



8.1.7 Counterfactuals

Next, we apply each of the “counterfactuals with reinforcement learning”, “counterfactual instances”, “contrastive explanation method”, and the “counterfactuals with prototypes” methods. We also plot the kernel SHAP values to show how the counterfactual methods change the attribution of each feature leading to the change in prediction.

Counter Factuals with Reinforcement Learning

CFRL trains a new model when fitting the explainer called an actor that takes instances and produces counterfactuals. It does this using reinforcement learning. In reinforcement learning, an actor model takes some state as input and generates actions; in our case, the actor takes an instance with a target classification and attempts to produce a member of the target class. Outcomes of actions are assigned rewards dependent on a reward function designed to encourage specific behaviors. In our case, we reward correctly classified counterfactuals generated by the actor. As well as this, we reward counterfactuals that are close to the data distribution as modeled by an autoencoder. Finally, we require that they are sparse perturbations of the original instance. The reinforcement training step pushes the actor to take high reward actions. CFRL is a black-box method as the process by which we update the actor to maximize the reward only requires estimating the reward via sampling the counterfactuals.

```
[21]: from alibi.explainers import CounterfactualRL

predict_fn = lambda x: model(x)
```

(continues on next page)

(continued from previous page)

```

cfrl_explainer = CounterfactualRL(
    predictor=predict_fn,          # The model to explain
    encoder=ae.encoder,           # The encoder
    decoder=ae.decoder,          # The decoder
    latent_dim=7,                 # The dimension of the autoencoder latent space
    coeff_sparsity=0.5,           # The coefficient of sparsity
    coeff_consistency=0.5,        # The coefficient of consistency
    train_steps=10000,           # The number of training steps
    batch_size=100,              # The batch size
)

cfrl_explainer.fit(X=scaler.transform(X_train))

result_cfrl = cfrl_explainer.explain(X=scaler.transform(x), Y_t=np.array([1]))
print("Instance class prediction:", model.predict(scaler.transform(x))[0].argmax())
print("Counterfactual class prediction:", model.predict(result_cfrl.data['cf']['X'])[0].
      ↪argmax())

```

100%|=====| 10000/10000 [01:50<00:00, 90.18it/s]
 100%|=====| 1/1 [00:00<00:00, 167.78it/s]

Instance class prediction: 0
 Counterfactual class prediction: 1

```

[22]: cfrl = scaler.inverse_transform(result_cfrl.data['cf']['X'])
      compare_instances(x, cfrl)

```

fixed acidity	instance:	9.2	counter factual:	8.965	difference: 0.
↪2350657					
volatile acidity	instance:	0.36	counter factual:	0.349	difference: 0.
↪0108247					
citric acid	instance:	0.34	counter factual:	0.242	difference: 0.
↪0977357					
residual sugar	instance:	1.6	counter factual:	2.194	difference: -
↪0.5943643					
chlorides	instance:	0.062	counter factual:	0.059	difference: 0.
↪0031443					
free sulfur dioxide	instance:	5.0	counter factual:	6.331	difference: -
↪1.3312454					
total sulfur dioxide	instance:	12.0	counter factual:	14.989	difference: -
↪2.9894428					
density	instance:	0.997	counter factual:	0.997	difference: -
↪0.0003435					
pH	instance:	3.2	counter factual:	3.188	difference: 0.
↪0118126					
sulphates	instance:	0.67	counter factual:	0.598	difference: 0.
↪0718592					
alcohol	instance:	10.5	counter factual:	9.829	difference: 0.
↪6712008					

```

[23]: from alibi.explainers import KernelShap

      predict_fn = lambda x: model(scaler.transform(x))

```

(continues on next page)

(continued from previous page)

```
explainer = KernelShap(predict_fn, task='classification')
explainer.fit(X_train[0:100])

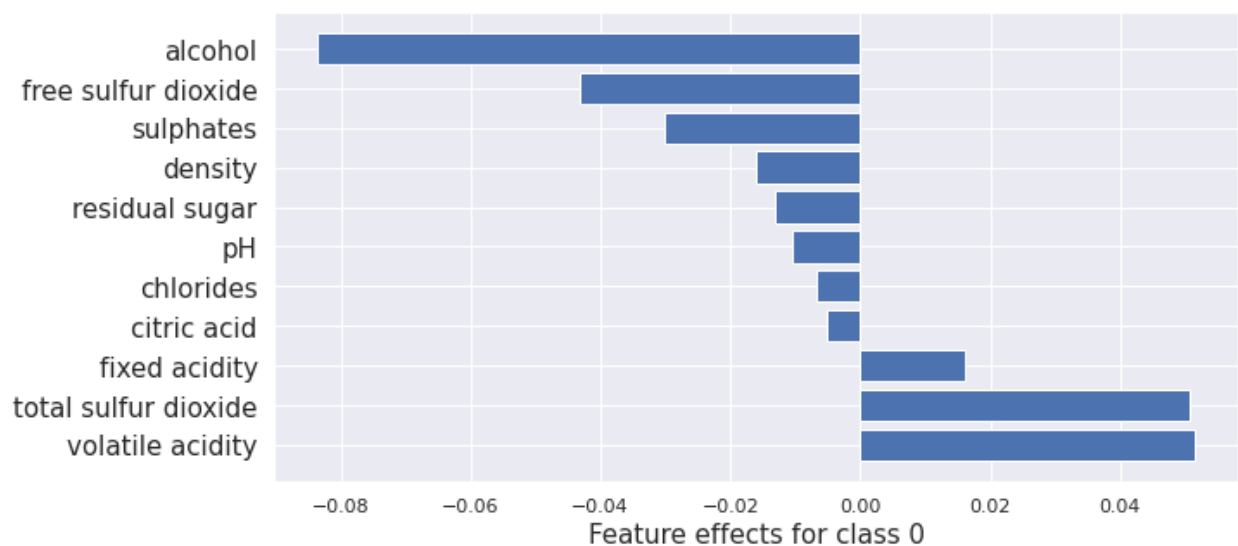
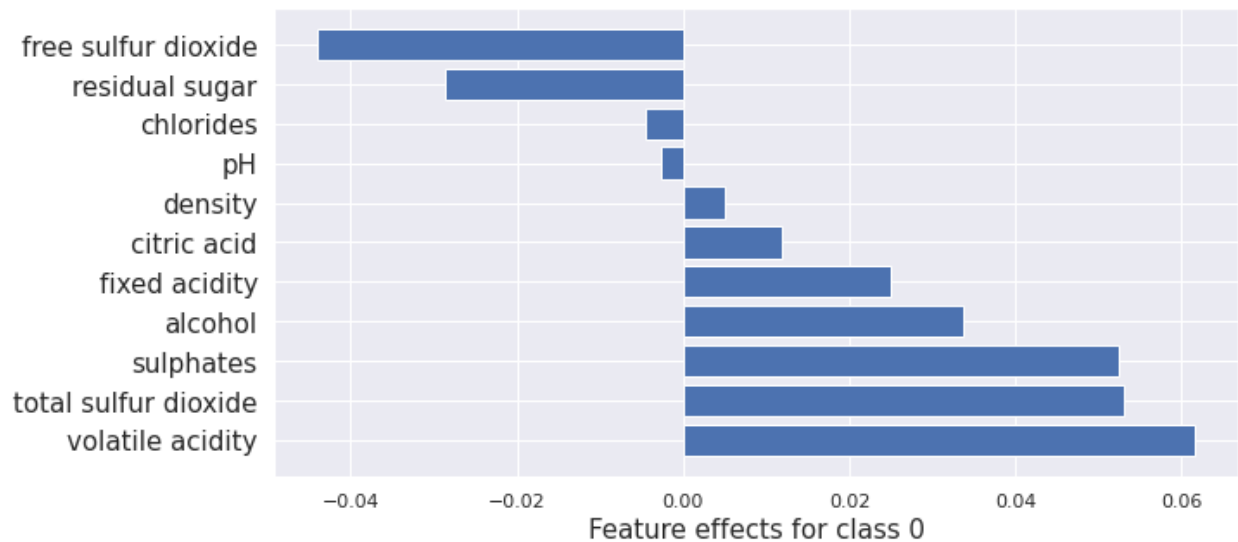
result_x = explainer.explain(x)
result_cfrl = explainer.explain(cfrl)

plot_importance(result_x.shap_values[0], features, 0)
plot_importance(result_cfrl.shap_values[0], features, 0)
```

```
100%|=====| 1/1 [00:00<00:00, 2.66it/s]
```

```
100%|=====| 1/1 [00:00<00:00, 2.55it/s]
```

[23]: (<AxesSubplot:xlabel='Feature effects for class 0'>,
<Figure size 720x360 with 1 Axes>)



Counterfactual Instances

First we need to revert to using tfv1

```
[24]: import tensorflow.compat.v1 as tf
      tf.disable_v2_behavior()
```

```
WARNING:tensorflow:From /home/alex/miniconda3/envs/alibi-explain/lib/python3.8/site-
packages/tensorflow/python/compat/v2_compat.py:111: disable_resource_variables (from
tensorflow.python.ops.variable_scope) is deprecated and will be removed in a future
version.
Instructions for updating:
non-resource variables are not supported in the long term
```

```
[25]: from tensorflow.keras.models import Model, load_model
      model = load_tf_model()
      ae = load_ae_model()
```

```
WARNING:tensorflow:OMP_NUM_THREADS is no longer used by the default Keras config. To
configure the number of threads, use tf.config.threading APIs.
WARNING:tensorflow:No training configuration found in the save file, so the model was
*not* compiled. Compile it manually.
WARNING:tensorflow:No training configuration found in the save file, so the model was
*not* compiled. Compile it manually.
```

The counterfactual instance method in alibi generates counterfactuals by defining a loss that prefers interpretable instances close to the target class. It then uses gradient descent to move within the feature space until it obtains a counterfactual of sufficient quality.

```
[26]: from alibi.explainers import Counterfactual
```

```
explainer = Counterfactual(
    model,                                # The model to explain
    shape=(1,) + X_train.shape[1:],      # The shape of the model input
    target_proba=0.51,                    # The target class probability
    tol=0.01,                             # The tolerance for the loss
    target_class='other',                 # The target class to obtain
)

result_cf = explainer.explain(scaler.transform(x))
print("Instance class prediction:", model.predict(scaler.transform(x))[0].argmax())
print("Counterfactual class prediction:", model.predict(result_cf.data['cf']['X'])[0].
      argmax())
```

```
WARNING:tensorflow:From /home/alex/Development/alibi-explain/alibi/explainers/
counterfactual.py:170: The name tf.keras.backend.get_session is deprecated. Please use
tf.compat.v1.keras.backend.get_session instead.
```

```
`Model.state_updates` will be removed in a future version. This property should not be
used in TensorFlow 2.0, as `updates` are applied automatically.
```

```
Instance class prediction: 0
Counterfactual class prediction: 1
```

```
[27]: cf = scaler.inverse_transform(result_cf.data['cf']['X'])
      compare_instances(x, cf)
```

fixed acidity ↪0.030319	instance:	9.2	counter factual:	9.23	difference: -
volatile acidity ↪0004017	instance:	0.36	counter factual:	0.36	difference: 0.
citric acid ↪0064294	instance:	0.34	counter factual:	0.334	difference: 0.
residual sugar ↪0179322	instance:	1.6	counter factual:	1.582	difference: 0.
chlorides ↪0011683	instance:	0.062	counter factual:	0.061	difference: 0.
free sulfur dioxide ↪0449123	instance:	5.0	counter factual:	4.955	difference: 0.
total sulfur dioxide ↪6759205	instance:	12.0	counter factual:	11.324	difference: 0.
density ↪5.08e-05	instance:	0.997	counter factual:	0.997	difference: -
pH ↪0012383	instance:	3.2	counter factual:	3.199	difference: 0.
sulphates ↪0297857	instance:	0.67	counter factual:	0.64	difference: 0.
alcohol ↪6195097	instance:	10.5	counter factual:	9.88	difference: 0.

```
[28]: from alibi.explainers import KernelShap

      predict_fn = lambda x: model.predict(scaler.transform(x))

      explainer = KernelShap(predict_fn, task='classification')

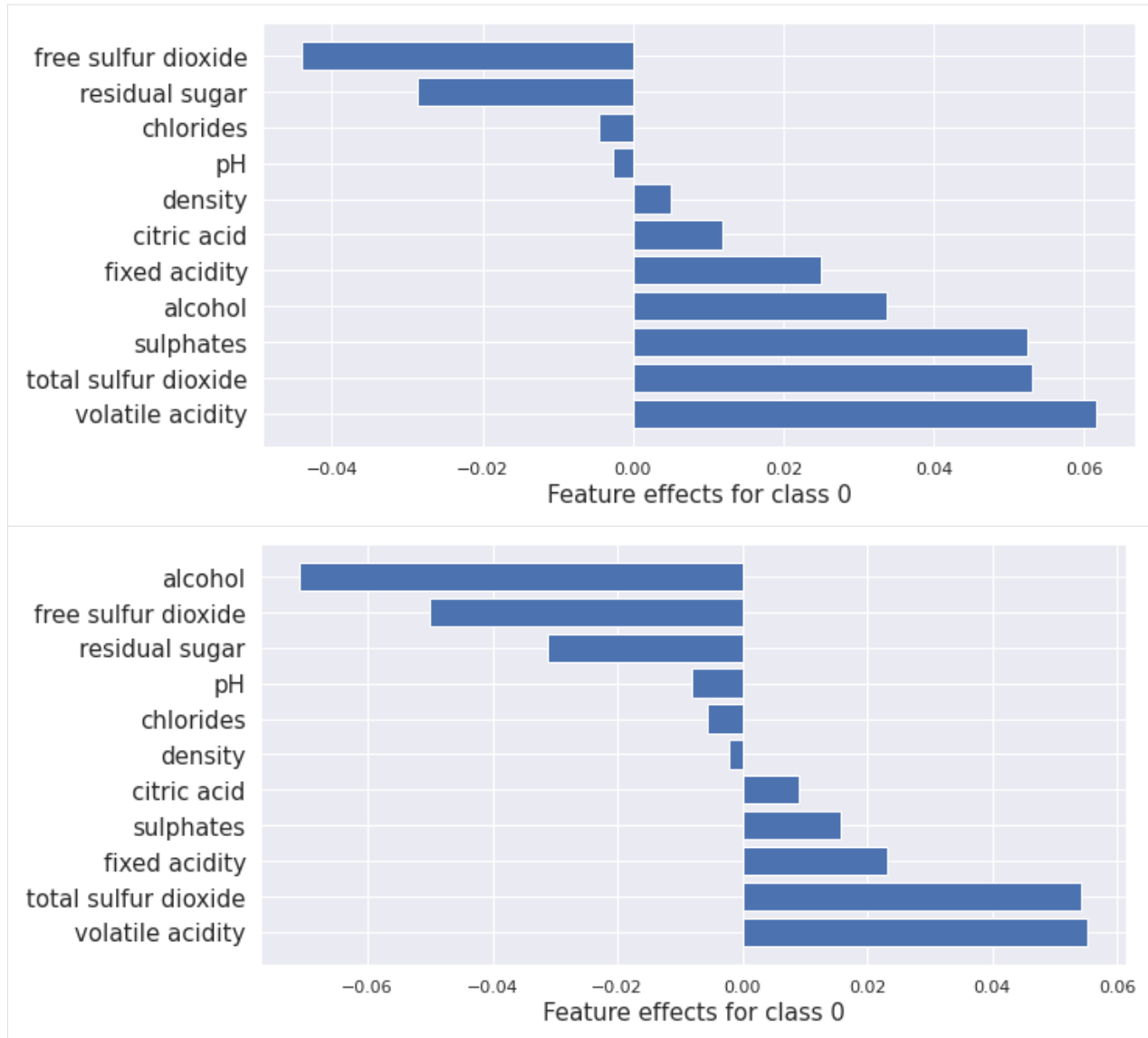
      explainer.fit(X_train[0:100])

      result_x = explainer.explain(x)
      result_cf = explainer.explain(cf)

      plot_importance(result_x.shap_values[0], features, 0)
      plot_importance(result_cf.shap_values[0], features, 0)
```

```
100%|=====| 1/1 [00:01<00:00, 1.19s/it]
100%|=====| 1/1 [00:01<00:00, 1.23s/it]
```

```
[28]: (<AxesSubplot:xlabel='Feature effects for class 0'>,
      <Figure size 720x360 with 1 Axes>)
```

Contrastive Explanations Method

The CEM method generates counterfactuals by defining a loss that prefers interpretable instances close to the target class. It also adds an autoencoder reconstruction loss to ensure the counterfactual stays within the data distribution.

```
[29]: from alibi.explainers import CEM

cem = CEM(model,
           shape=(1,) + X_train.shape[1:],
           mode='PN',
           kappa=0.2,
           beta=0.1,
           ae_model=ae)

# model to explain
# shape of the model input
# pertinent negative mode
# Confidence parameter for the attack loss
# Regularization constant for L1 loss term
# autoencoder model
```

(continues on next page)

(continued from previous page)

```

cem.fit(
    scaler.transform(X_train), # scaled training data
    no_info_type='median'      # non-informative value for each feature
)
result_cem = cem.explain(scaler.transform(x), verbose=False)
cem_cf = result_cem.data['PN']

print("Instance class prediction:", model.predict(scaler.transform(x))[0].argmax())
print("Counterfactual class prediction:", model.predict(cem_cf)[0].argmax())

Instance class prediction: 0
Counterfactual class prediction: 1

```

```

[30]: cem_cf = result_cem.data['PN']
cem_cf = scaler.inverse_transform(cem_cf)
compare_instances(x, cem_cf)

```

fixed acidity ↪ 2e-07	instance:	9.2	counter factual:	9.2	difference: ↪
volatile acidity ↪ 0.0	instance:	0.36	counter factual:	0.36	difference: -
citric acid ↪ 0.0	instance:	0.34	counter factual:	0.34	difference: -
residual sugar ↪ 1211611	instance:	1.6	counter factual:	1.479	difference: 0.
chlorides ↪ 0045941	instance:	0.062	counter factual:	0.057	difference: 0.
free sulfur dioxide ↪ 2929246	instance:	5.0	counter factual:	2.707	difference: 2.
total sulfur dioxide ↪ 9e-06	instance:	12.0	counter factual:	12.0	difference: 1.
density ↪ 0.0004602	instance:	0.997	counter factual:	0.997	difference: -
pH ↪ 0.0	instance:	3.2	counter factual:	3.2	difference: -
sulphates ↪ 121454	instance:	0.67	counter factual:	0.549	difference: 0.
alcohol ↪ 8478804	instance:	10.5	counter factual:	9.652	difference: 0.

```

[31]: from alibi.explainers import KernelShap

predict_fn = lambda x: model.predict(scaler.transform(x))

explainer = KernelShap(predict_fn, task='classification')

explainer.fit(X_train[0:100])

result_x = explainer.explain(x)
result_cem_cf = explainer.explain(cem_cf)

```

(continues on next page)

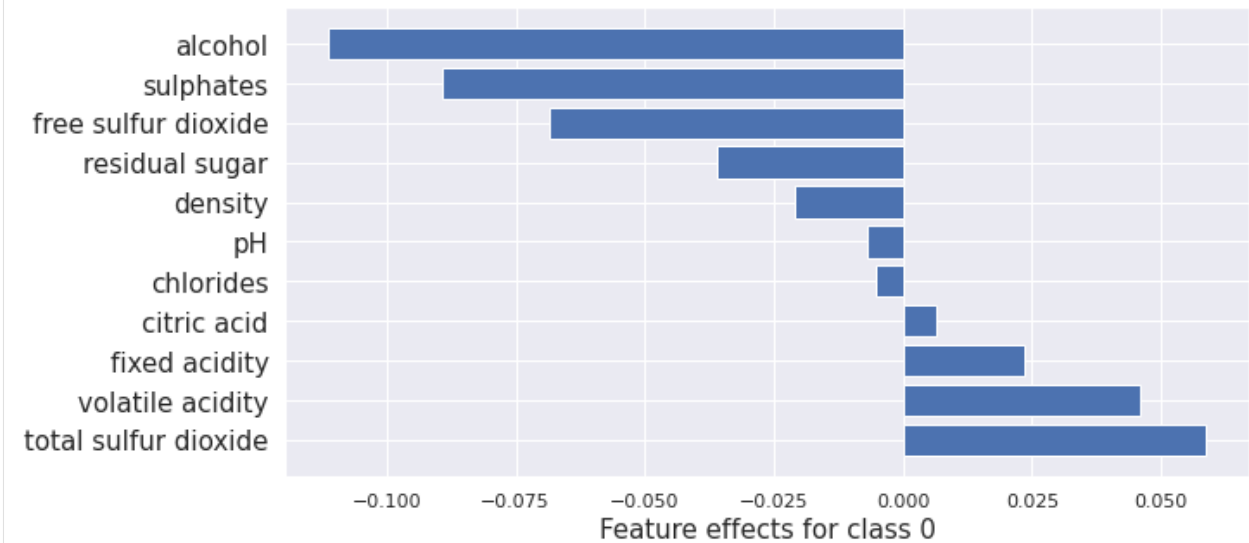
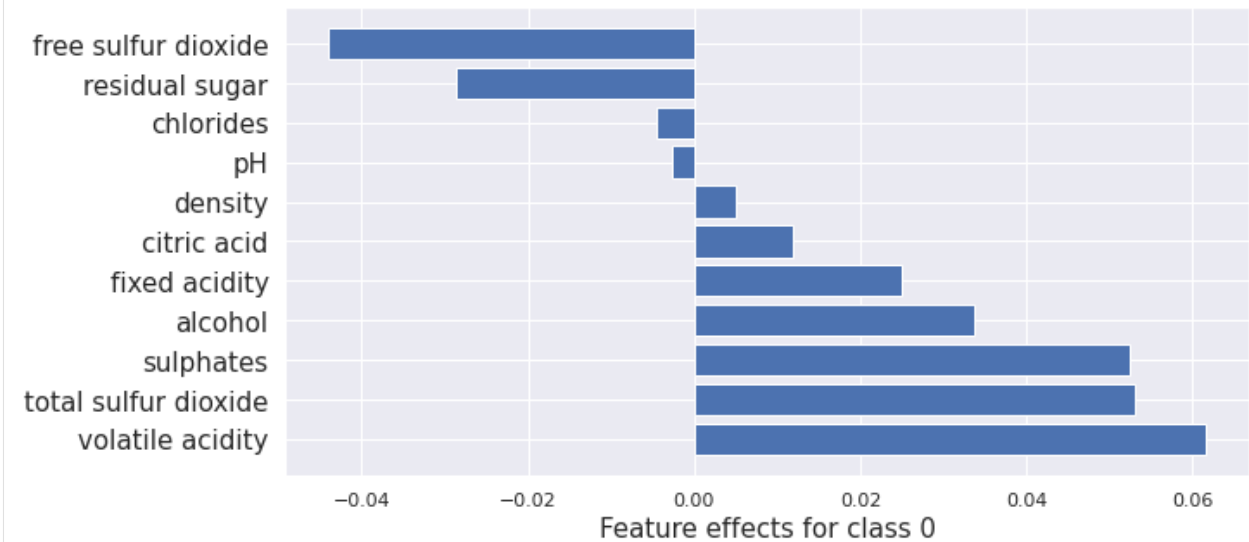
(continued from previous page)

```
plot_importance(result_x.shap_values[0], features, 0)
plot_importance(result_cem_cf.shap_values[0], features, 0)
```

```
100%|=====| 1/1 [00:01<00:00, 1.22s/it]
```

```
100%|=====| 1/1 [00:01<00:00, 1.21s/it]
```

```
[31]: (<AxesSubplot:xlabel='Feature effects for class 0'>,
      <Figure size 720x360 with 1 Axes>)
```



Counterfactual With Prototypes

Like the previous two methods, “counterfactuals with prototypes” defines a loss that guides the counterfactual towards the target class while also using an autoencoder to ensure it stays within the data distribution. As well as this, it uses prototype instances of the target class to ensure that the generated counterfactual is interpretable as a member of the target class.

```
[32]: from alibi.explainers import CounterfactualProto

explainer = CounterfactualProto(
    model,                                # The model to explain
    shape=(1,) + X_train.shape[1:],      # shape of the model input
    ae_model=ae,                          # The autoencoder
    enc_model=ae.encoder                   # The encoder
)

explainer.fit(scaler.transform(X_train)) # Fit the explainer with scaled data

result_proto = explainer.explain(scaler.transform(x), verbose=False)

proto_cf = result_proto.data['cf']['X']
print("Instance class prediction:", model.predict(scaler.transform(x))[0].argmax())
print("Counterfactual class prediction:", model.predict(proto_cf)[0].argmax())

`Model.state_updates` will be removed in a future version. This property should not be
used in TensorFlow 2.0, as `updates` are applied automatically.
```

```
Instance class prediction: 0
Counterfactual class prediction: 1
```

```
[33]: proto_cf = scaler.inverse_transform(proto_cf)
      compare_instances(x, proto_cf)
```

fixed acidity ↪ 2e-07	instance:	9.2	counter factual:	9.2	difference: ↪
volatile acidity ↪ 0.0	instance:	0.36	counter factual:	0.36	difference: -
citric acid ↪ 0.0	instance:	0.34	counter factual:	0.34	difference: -
residual sugar ↪ 1e-07	instance:	1.6	counter factual:	1.6	difference: ↪
chlorides ↪ 0.0	instance:	0.062	counter factual:	0.062	difference: ↪
free sulfur dioxide ↪ 0.0	instance:	5.0	counter factual:	5.0	difference: ↪
total sulfur dioxide ↪ 9e-06	instance:	12.0	counter factual:	12.0	difference: 1.
density ↪ 0.0	instance:	0.997	counter factual:	0.997	difference: -
pH ↪ 0.0	instance:	3.2	counter factual:	3.2	difference: -
sulphates ↪ 0470144	instance:	0.67	counter factual:	0.623	difference: 0.
alcohol ↪ 558073	instance:	10.5	counter factual:	9.942	difference: 0.

```
[34]: from alibi.explainers import KernelShap

predict_fn = lambda x: model.predict(scaler.transform(x))

explainer = KernelShap(predict_fn, task='classification')

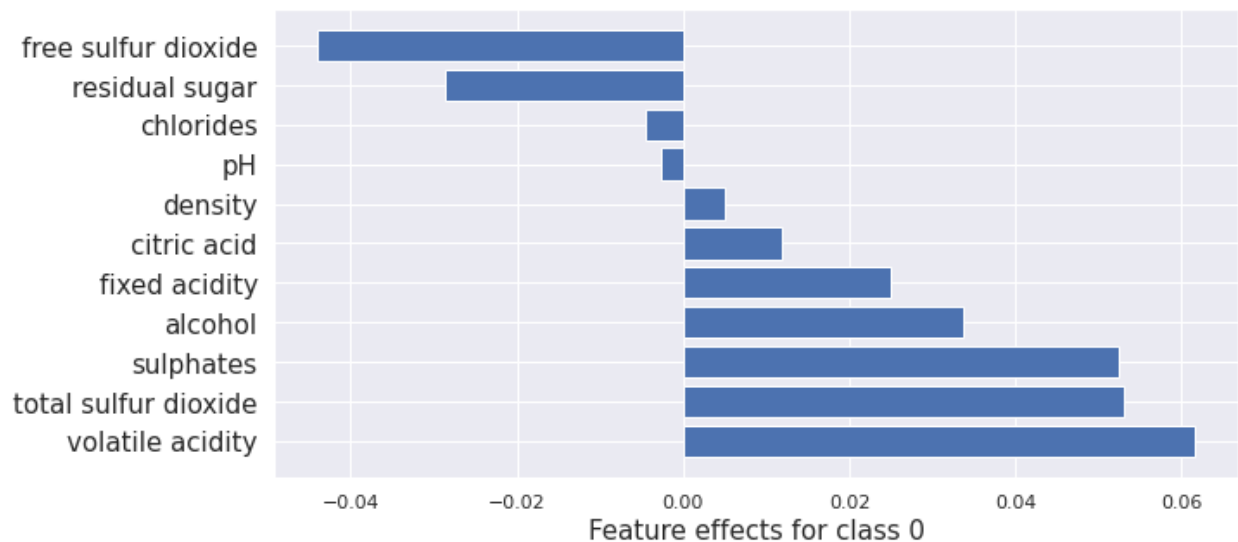
explainer.fit(X_train[0:100])

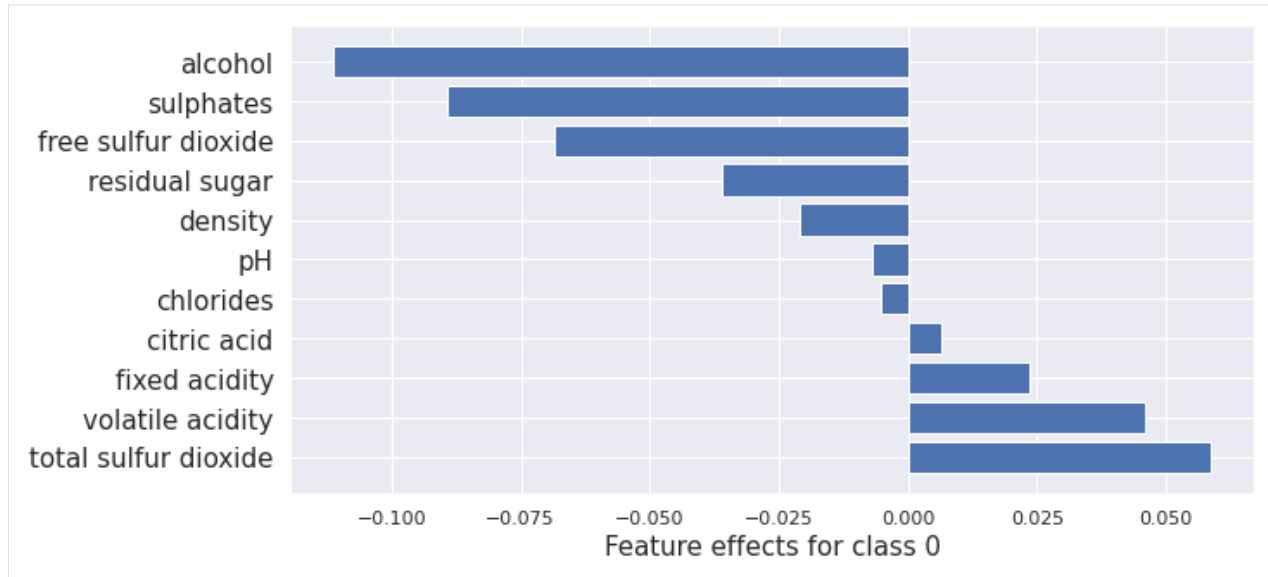
result_x = explainer.explain(x)
result_proto_cf = explainer.explain(cem_cf)

plot_importance(result_x.shap_values[0], features, 0)
print(result_x.shap_values[0].sum())
plot_importance(result_proto_cf.shap_values[0], features, 0)
print(result_proto_cf.shap_values[0].sum())

100%|=====| 1/1 [00:01<00:00, 1.20s/it]
100%|=====| 1/1 [00:01<00:00, 1.25s/it]
```

```
0.16304971136152735
-0.20360063157975683
```

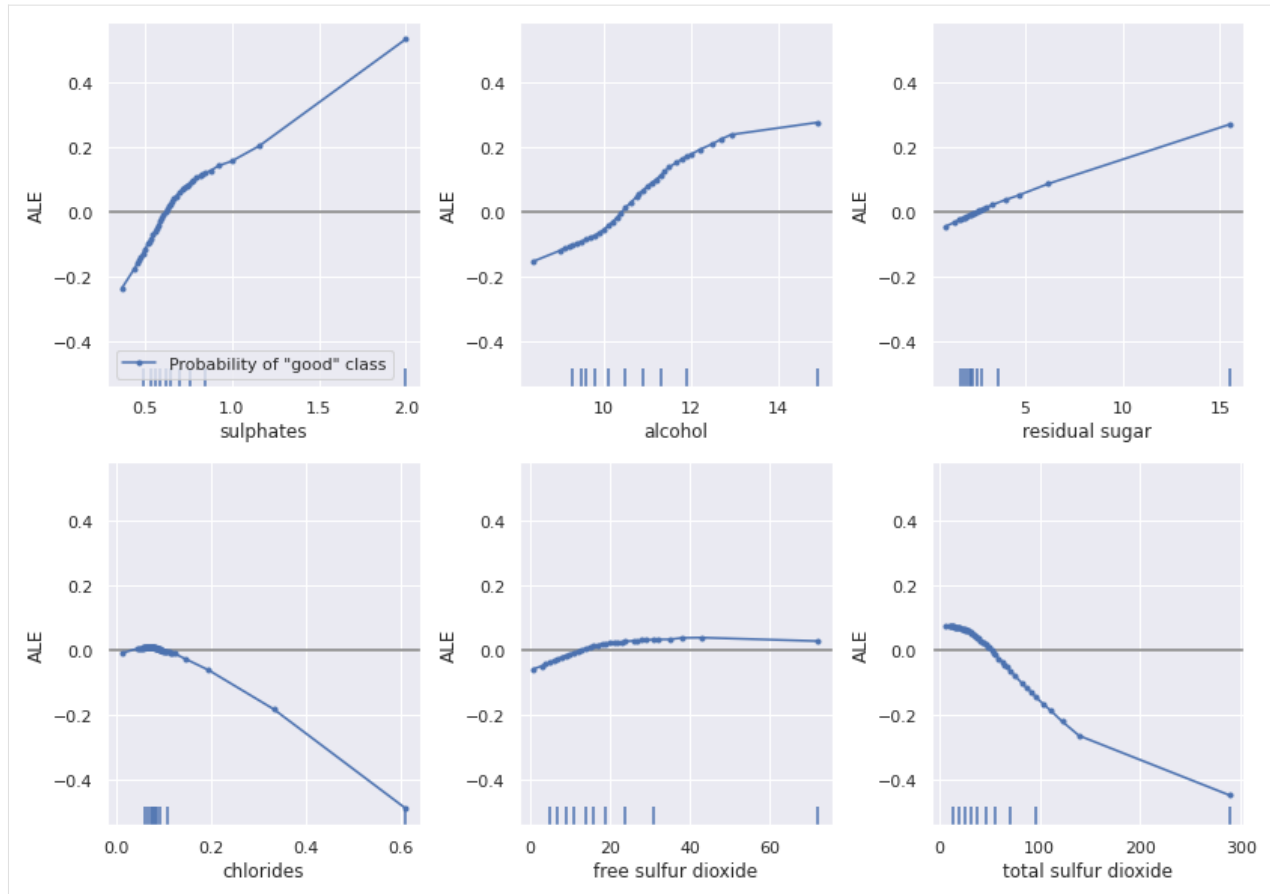




Looking at the ALE plots below, we can see how the counterfactual methods change the features to flip the prediction. Note that the ALE plots potentially miss details local to individual instances as they are global insights.

```
[35]: plot_ale(exp, features=['sulphates', 'alcohol', 'residual sugar', 'chlorides', 'free_
↳sulfur dioxide', 'total sulfur dioxide'], line_kw={'label': 'Probability of "good"
↳class'})
```

```
[35]: array([[<AxesSubplot:xlabel='sulphates', ylabel='ALE'>,
<AxesSubplot:xlabel='alcohol', ylabel='ALE'>,
<AxesSubplot:xlabel='residual sugar', ylabel='ALE'>],
[<AxesSubplot:xlabel='chlorides', ylabel='ALE'>,
<AxesSubplot:xlabel='free sulfur dioxide', ylabel='ALE'>,
<AxesSubplot:xlabel='total sulfur dioxide', ylabel='ALE'>]],
dtype=object)
```



8.2 Accumulated Local Effects

8.2.1 Accumulated Local Effects for classifying flowers

In this example we will explain the behaviour of classification models on the Iris dataset. It is recommended to first read the [ALE regression example](#) to familiarize yourself with how to interpret ALE plots in a simpler setting. Interpreting ALE plots for classification problems become more complex due to a few reasons:

- Instead of one ALE line for each feature we now have one for each class to explain the feature effects towards predicting each class.
- There are two ways to choose the prediction function to explain:
 - Class probability predictions (e.g. `clf.predict_proba` in sklearn)
 - Margin or logit predictions (e.g. `clf.decision_function` in sklearn)

We will see the implications of explaining each of these prediction functions.

```
[1]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
```

(continues on next page)

(continued from previous page)

```
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from alibi.explainers import ALE, plot_ale
```

Load and prepare the dataset

```
[2]: data = load_iris()
feature_names = data.feature_names
target_names = data.target_names
X = data.data
y = data.target
print(feature_names)
print(target_names)

['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
['setosa' 'versicolor' 'virginica']
```

Shuffle the data and define the train and test set:

```
[3]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_
↪state=42)
```

Fit and evaluate a logistic regression model

```
[4]: lr = LogisticRegression(max_iter=200)

[5]: lr.fit(X_train, y_train)
[5]: LogisticRegression(max_iter=200)

[6]: accuracy_score(y_test, lr.predict(X_test))
[6]: 1.0
```

Calculate Accumulated Local Effects

There are several options for explaining the classifier predictions using ALE. We define two prediction functions, one in the unnormalized logit space and the other in probability space, and look at how the resulting ALE plot interpretation changes.

```
[7]: logit_fun_lr = lr.decision_function
proba_fun_lr = lr.predict_proba

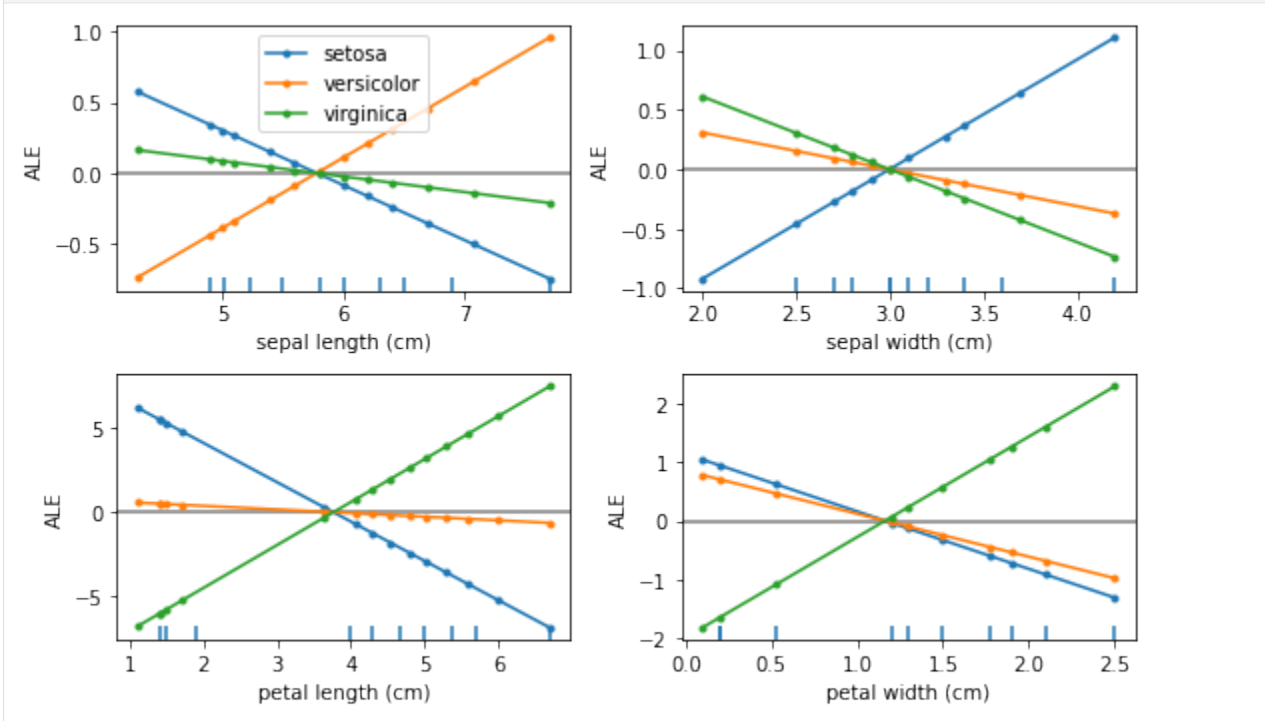
[8]: logit_ale_lr = ALE(logit_fun_lr, feature_names=feature_names, target_names=target_names)
proba_ale_lr = ALE(proba_fun_lr, feature_names=feature_names, target_names=target_names)

[9]: logit_exp_lr = logit_ale_lr.explain(X_train)
proba_exp_lr = proba_ale_lr.explain(X_train)
```


ALE in logit space

We first look at the ALE plots for explaining the feature effects towards the unnormalized logit scores:

```
[10]: plot_ale(logit_exp_lr, n_cols=2, fig_kw={'figwidth': 8, 'figheight': 5}, sharey=None);
```

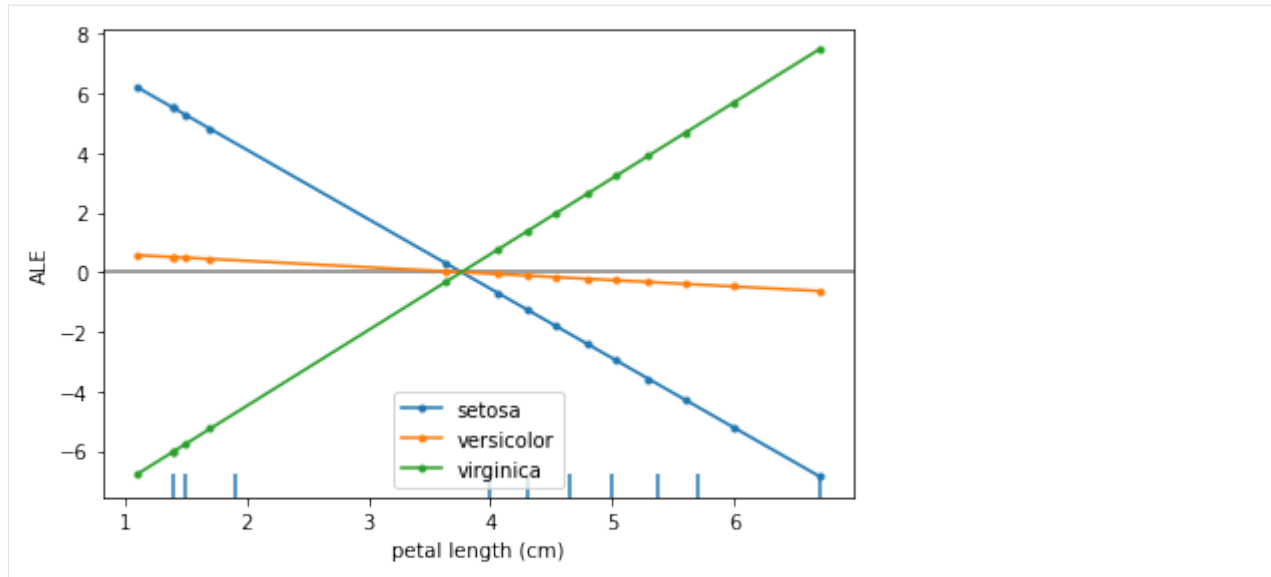


We see that the feature effects are linear for each class and each feature. This is exactly what we expect because the logistic regression is a linear model in the logit space.

Furthermore, the units of the ALE plots here are in logits, which means that the feature effect at some feature value will be a positive or negative contribution to the logit of each class with respect to the mean feature effect.

Let's look at the interpretation of the feature effects for "petal length" in more detail:

```
[11]: plot_ale(logit_exp_lr, features=[2]);
```



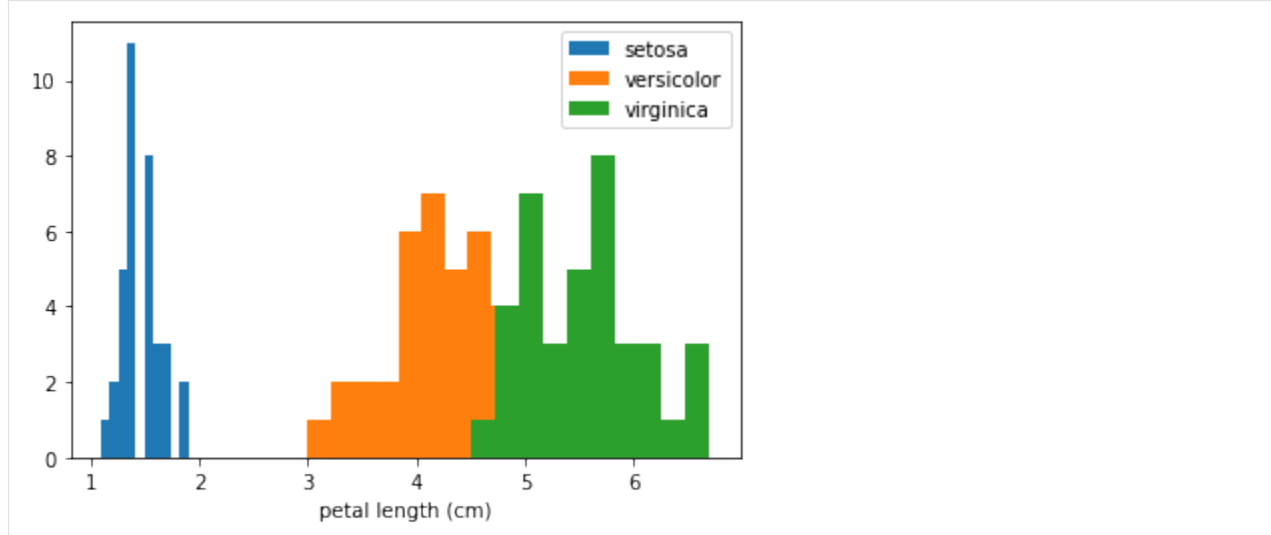
The main insights from an ALE plot are qualitative—we can make several observations:

- The slope of each ALE curve determines the relative effect of the feature `petal length` on the prediction (in logits) for each target class
- In particular, we observe that the feature `petal length` has little relative variation in its effect towards the target class of `versicolor`
- On the other hand, for the target classes of `setosa` and `virginica` the slopes of the curves are significant—the relative feature effect of `petal length` rises/falls for the target class of `virginica`/`setosa` as the `petal length` increases
- The effect of `petal length` on the target classes of `setosa` and `virginica` are inversely related, suggesting that e.g. the effect of longer petal lengths contributes more positively towards predicting `virginica` and negatively towards predicting `setosa`

We can gain even more insight into the ALE plot by looking at the class histograms for the feature `petal length`:

```
[12]: fig, ax = plt.subplots()
      for target in range(3):
          ax.hist(X_train[y_train==target][:,2], label=target_names[target]);

      ax.set_xlabel(feature_names[2])
      ax.legend();
```



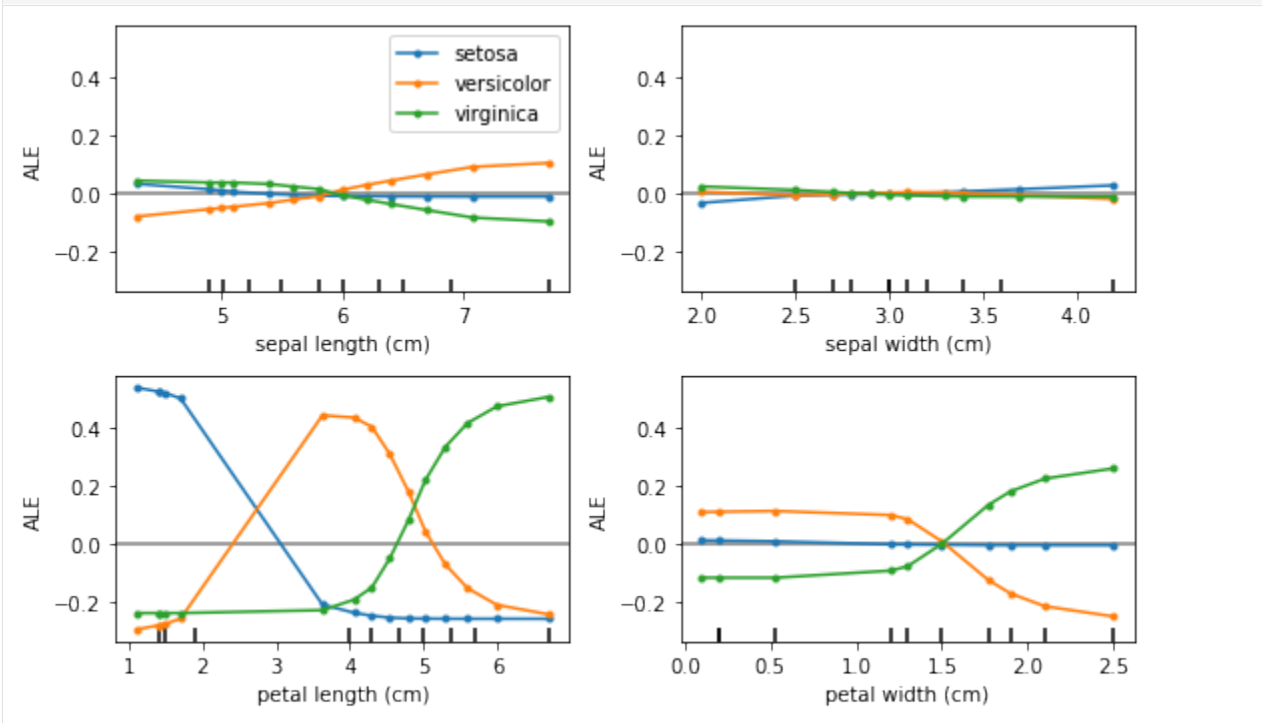
Here we see that the three classes are very well separated by this feature. This confirms that the ALE plot is behaving

as expected—the feature effects of small value of `petal length` are that of increasing the logit values for the class `setosa` and decreasing for the other two classes. Also note that the range of the ALE values for this feature is particularly high compared to other features which can be interpreted as the model attributing more importance to this feature as it separates the classes well on its own.

ALE in probability space

We now turn to interpret the ALE plots for explaining the feature effects on the probabilities of each class.

```
[17]: plot_ale(proba_exp_lr, n_cols=2, fig_kw={'figwidth': 8, 'figheight': 5});
```



As expected, the ALE plots are no longer linear which reflects the non-linear nature due to the softmax transformation applied to the logits.

Note that, in this case, the ALE are in the units of relative probability mass, i.e. given a feature value how much more (less) probability does the model assign to each class relative to the mean effect of that feature. This also means that any increase in relative probability of one class must result in a decrease in probability of another class. In fact, the ALE curves summed across classes result in 0 as a direct consequence of conservation of probability:

```
[18]: for feature in range(4):
      print(proba_exp_lr.ale_values[feature].sum())
```

```
-5.551115123125783e-17
1.734723475976807e-17
-6.661338147750939e-16
4.440892098500626e-16
```

By transforming the ALE plots into probability space we can gain additional insight into the model behaviour. For example, the ALE curve for the feature `petal width` and class `setosa` is virtually flat. This means that the model does not use this feature to assign higher or lower probability to class `setosa` with respect to the average effect of that feature. This is not readily seen in logit space as the ALE curve has negative slope which would lead us to the opposite conclusion. The interpretation here is that even though the ALE curve in the logit space shows a negative effect with

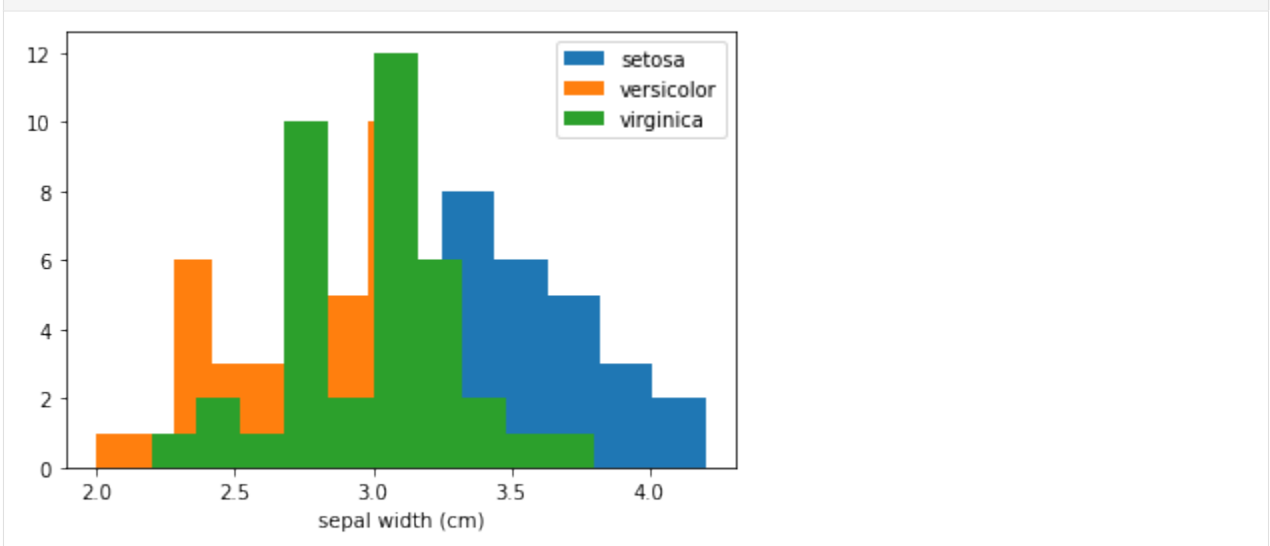
feature value, the effect in the logit space is not significant enough to translate into a tangible effect in the probability space.

Turning to the feature `petal length` we can observe a much more nuanced behaviour of the ALE plots than we saw in the logit space previously. In particular, for the target class `versicolor`, whilst the ALE curve is nearly flat in the logit space, in probability space it reveals a significant uplift over the average effect of `petal length` towards predicting `versicolor` in an interval between ~3–5cm. This agrees with our observation previously that the histogram of `petal length` by target class reveals that the feature can separate all three classes quite well.

Finally, the feature `sepal width` does not offer significant information to the model to prefer any class over the other (with respect to the mean effect of `sepal_width` that is). If we plot the marginal distribution of `sepal_width` it explains why that is—the overlap in the class conditional histograms of this feature show that it does not increase the model discriminative power:

```
[14]: fig, ax = plt.subplots()
      for target in range(3):
          ax.hist(X_train[y_train==target][:,1], label=target_names[target]);

      ax.set_xlabel(feature_names[1])
      ax.legend();
```



ALE for gradient boosting

Finally, we look at the resulting ALE plots for a highly non-linear model—a gradient boosted classifier.

```
[15]: from sklearn.ensemble import GradientBoostingClassifier
```

```
[16]: gb = GradientBoostingClassifier()
      gb.fit(X_train, y_train)
```

```
[16]: GradientBoostingClassifier()
```

```
[17]: accuracy_score(y_test, gb.predict(X_test))
```

```
[17]: 1.0
```

As before, we explain the feature contributions in both logit and probability space.

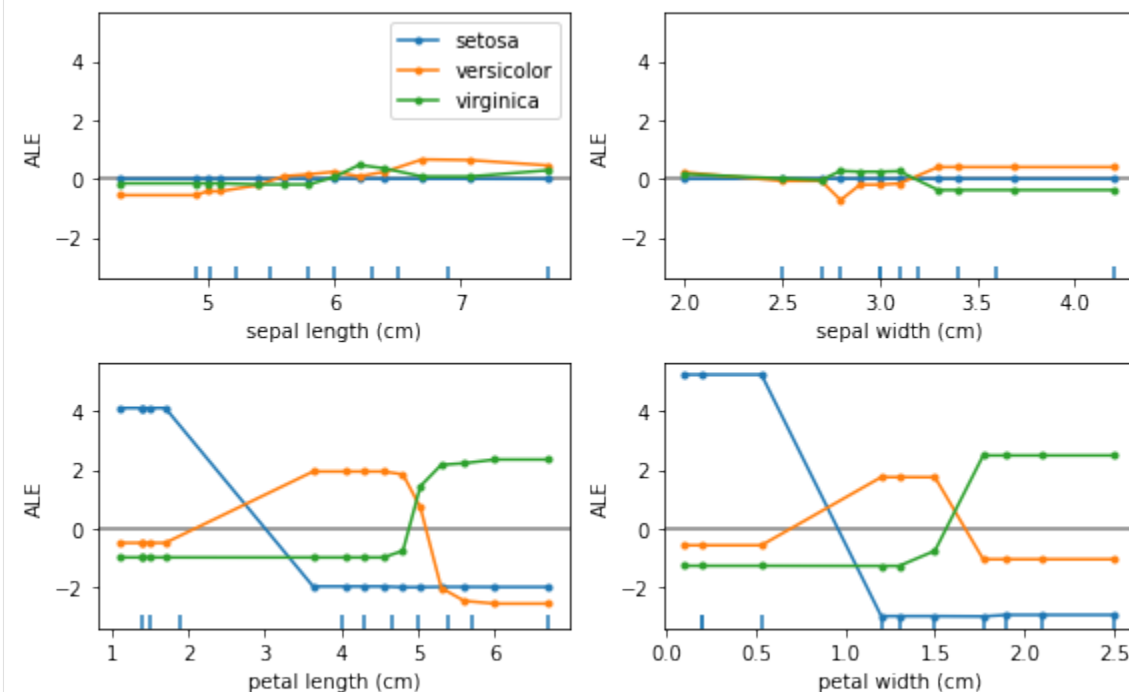
```
[18]: logit_fun_gb = gb.decision_function
      proba_fun_gb = gb.predict_proba
```

```
[19]: logit_ale_gb = ALE(logit_fun_gb, feature_names=feature_names, target_names=target_names)
      proba_ale_gb = ALE(proba_fun_gb, feature_names=feature_names, target_names=target_names)
```

```
[20]: logit_exp_gb = logit_ale_gb.explain(X_train)
      proba_exp_gb = proba_ale_gb.explain(X_train)
```

ALE in logit space

```
[21]: plot_ale(logit_exp_gb, n_cols=2, fig_kw={'figwidth': 8, 'figheight': 5});
```



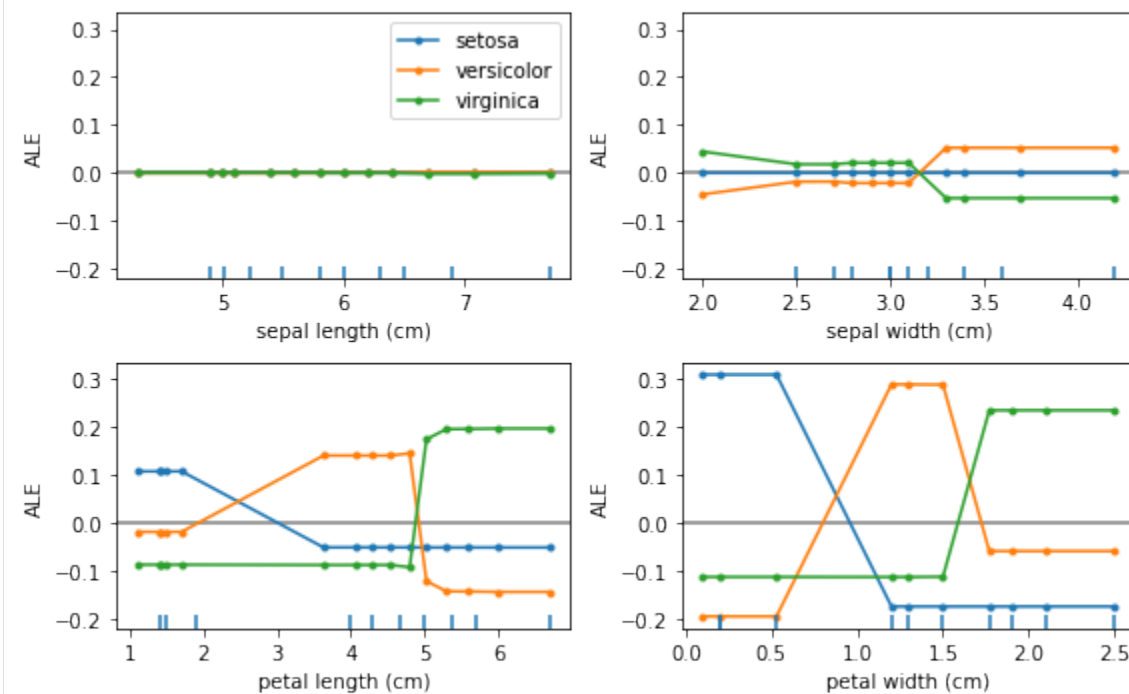
The ALE curves are no longer linear as the model used is non-linear. Furthermore, we've plotted the ALE curves of different features on the same scale on the y -axis which suggests that the features `petal length` and `petal width` are more discriminative for the task. Checking the feature importances of the classifier confirms this:

```
[22]: gb.feature_importances_
```

```
[22]: array([0.00221272, 0.01651258, 0.51811252, 0.46316218])
```

ALE in probability space

```
[23]: plot_ale(proba_exp_gb, n_cols=2, fig_kw={'figwidth': 8, 'figheight': 5});
```

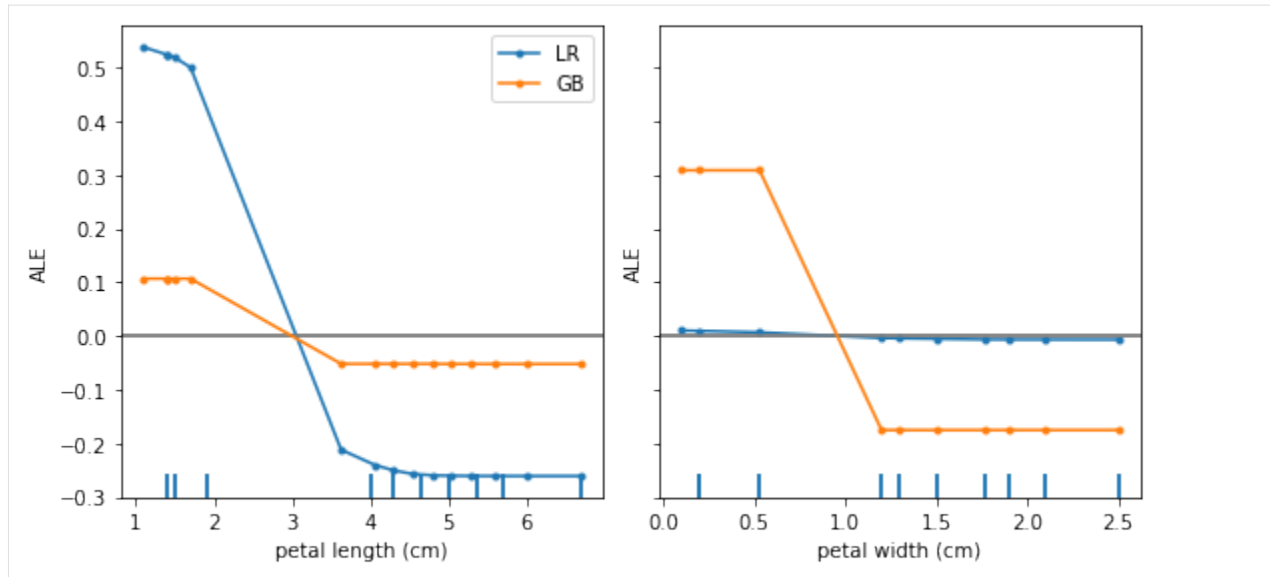


Because of the non-linearity of the gradient boosted model the ALE curves in probability space are very similar to the curves in the logit space just on a different scale.

Comparing ALE between models

We have seen that for both logistic regression and gradient boosting models the features `petal length` and `petal width` have a high feature effect on the classifier predictions. We can explore this in more detail by comparing the ALE curves for both models. In the following we plot the ALE curves of the two features for predicting the class `setosa` in probability space:

```
[24]: fig, ax = plt.subplots(1, 2, figsize=(8, 4), sharey='row');
plot_ale(proba_exp_lr, features=[2, 3], targets=['setosa'], ax=ax, line_kw={'label': 'LR',
↪});
plot_ale(proba_exp_gb, features=[2, 3], targets=['setosa'], ax=ax, line_kw={'label': 'GB',
↪});
```



From this plot we can draw a couple of conclusions:

- Both models have similar feature effects of `petal length`—a high positive effect for predicting `setosa` for small feature values and a high negative effect for large values (over >3cm).
- While the logistic regression model does not benefit much from the `petal width` feature to discriminate the `setosa` class, the gradient boosted model does exploit this feature to discern between different classes.

8.2.2 Accumulated Local Effects for predicting house prices

In this example we will explain the behaviour of regression models on the California housing dataset. We will show how to calculate accumulated local effects (ALE) for determining the feature effects on a model and how these vary on different kinds of models (linear and non-linear models).

```
[1]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from alibi.explainers import ALE, plot_ale
```

Fetch and prepare the dataset

Each row in the dataset represents a whole census block (the smallest geographical unit for which the US census publishes data), thus the feature values for each datapoint are averages within the block. For a complete description of the dataset please refer to the [scikit-learn documentation page](#).

```
[2]: data = fetch_california_housing(as_frame=True)
feature_names = data.feature_names
```

```
[3]: data.frame.head()
```

```
[3]:   MedInc   HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0   8.3252    41.0   6.984127   1.023810    322.0   2.555556    37.88
1   8.3014    21.0   6.238137   0.971880    2401.0   2.109842    37.86
2   7.2574    52.0   8.288136   1.073446    496.0   2.802260    37.85
3   5.6431    52.0   5.817352   1.073059    558.0   2.547945    37.85
4   3.8462    52.0   6.281853   1.081081    565.0   2.181467    37.85

   Longitude  MedHouseVal
0   -122.23      4.526
1   -122.22      3.585
2   -122.24      3.521
3   -122.25      3.413
4   -122.25      3.422
```

```
[4]: X, y = data.data.to_numpy(), data.target.to_numpy()
```

Shuffle the data and define the train and test set:

```
[5]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_
↪state=42)
```

Fit and evaluate models

Fit and evaluate a linear regression model:

```
[6]: lr = LinearRegression()
```

```
[7]: lr.fit(X_train, y_train)
```

```
[7]: LinearRegression()
```

```
[8]: mean_squared_error(y_test, lr.predict(X_test))
```

```
[8]: 0.5411287478470682
```

Fit and evaluate a random forest model:

```
[9]: rf = RandomForestRegressor()
```

```
[10]: rf.fit(X_train, y_train)
```

```
[10]: RandomForestRegressor()
```

```
[11]: mean_squared_error(y_test, rf.predict(X_test))
```

```
[11]: 0.2538208780210583
```


Feature Effects: Motivation

Here we develop an intuition for calculating feature effects. We start by illustrating the calculation of feature effects for the linear regression model.

For our regression model, the conditional mean or the prediction function $\mathbb{E}(y|x) = f(x)$ is linear:

$$f(x) = w_0 + w_1x_1 + \dots + w_kx_k.$$

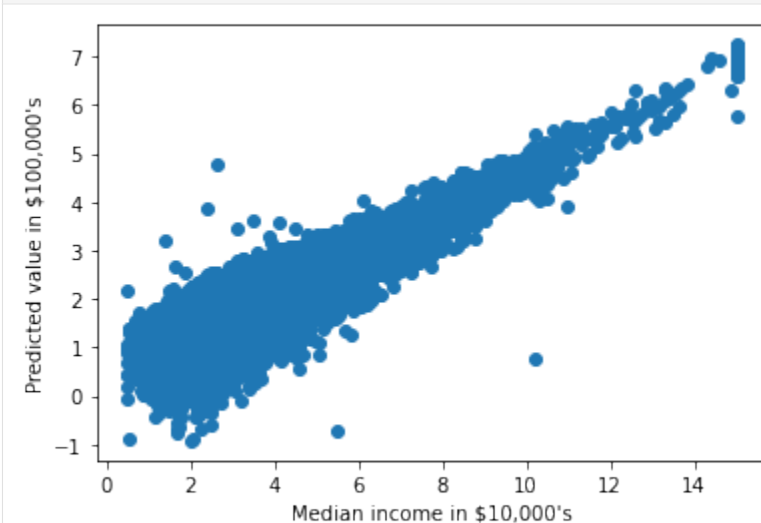
Because the model is additive and doesn't include feature interactions, we can read off individual feature effects immediately: the effect of any feature x_i is just w_ix_i , so the effect is a linear function of x_i and the sign of the coefficient w_i determines whether the effect is positive or negative as x_i changes.

Now suppose we don't know the true effect of the feature x_i which is usually the case when using a more complex model. How might we approach the problem of estimating the effect? Let's focus on one feature - median income (MedInc). The following is a scatterplot of model predictions versus the feature:

```
[12]: FEATURE = 'MedInc'
      index = feature_names.index(FEATURE)

      fig, ax = plt.subplots()
      ax.scatter(X_train[:, index], lr.predict(X_train));

      ax.set_xlabel('Median income in $10,000\'s');
      ax.set_ylabel('Predicted value in $100,000\'s');
```



As we can see, there is a strong positive correlation as one might expect. However the feature effects for **MedInc** cannot be read off immediately because the prediction function includes the effects of all features not just **MedInc**. What we need is a procedure to block out the effects of all other features to uncover the true effect of **MedInc** only. This is exactly what the ALE approach does by averaging the differences of predictions across small intervals of the feature.

Calculate Accumulated Local Effects

Here we initialize the ALE object by passing it the predictor function which in this case is the `clf.predict` method for both models. We also pass in feature names and target name for easier interpretation of the resulting explanations.

```
[13]: lr_ale = ALE(lr.predict, feature_names=feature_names, target_names=['Value in $100,000\'s
      ↪'])
      rf_ale = ALE(rf.predict, feature_names=feature_names, target_names=['Value in $100,000\'s
      ↪'])
```

We now call the `explain` method on the explainer objects which will compute the ALE's and return an `Explanation` object which is ready for inspection and plotting. Since ALE is a global explanation method it takes in a batch of data for which the model feature effects are computed, in this case we pass in the training set.

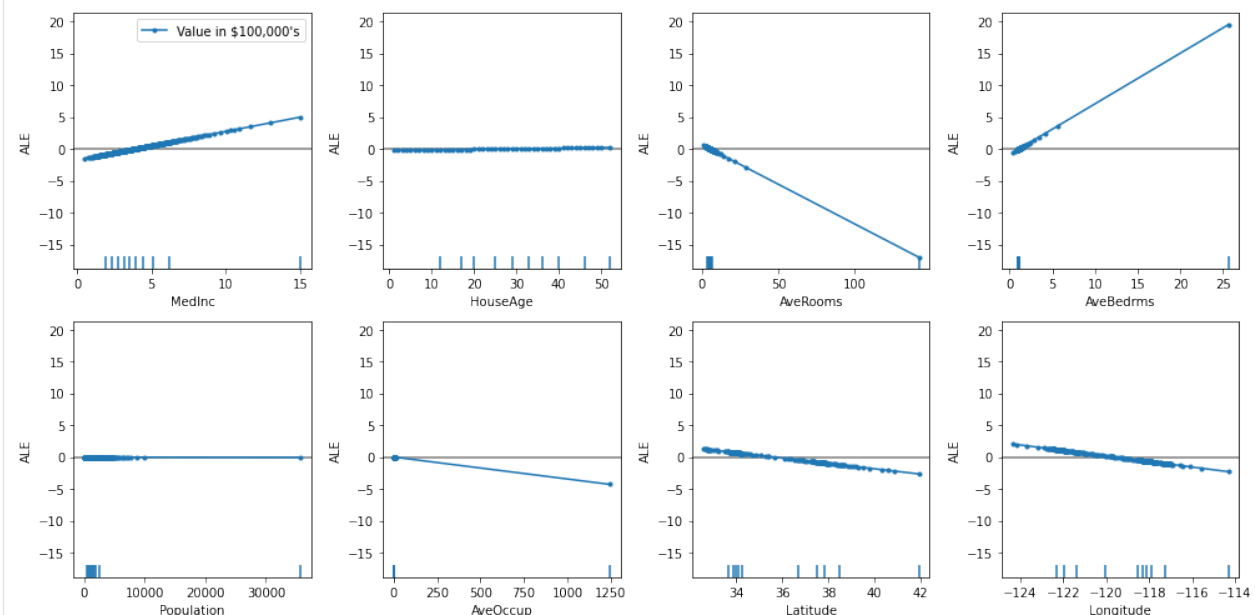
```
[14]: lr_exp = lr_ale.explain(X_train)
      rf_exp = rf_ale.explain(X_train)
```

The resulting `Explanation` objects contain the ALE's for each feature under the `ale_values` attribute - this is a list of numpy arrays, one for each feature. The easiest way to interpret the ALE values is by plotting them against the feature values for which we provide a built-in function `plot_ale`. By calling the function without arguments, we will plot the effects of every feature, so in this case we will get 8 different subplots. To fit them all on the screen we pass in options for the figure size.

ALE for the linear regression model

The ALE plots show the main effects of each feature on the prediction function.

```
[15]: plot_ale(lr_exp, n_cols=4, fig_kw={'figwidth':14, 'figheight': 7});
```

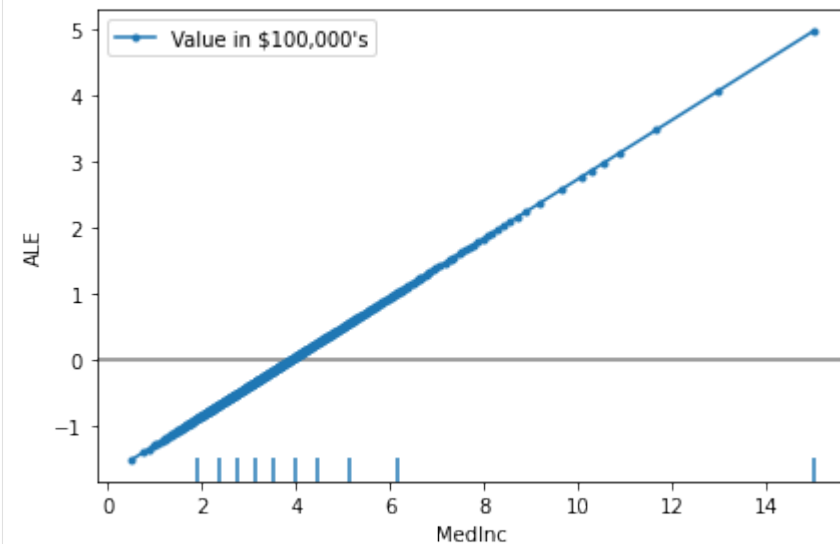


As expected, the feature effects plots are linear because we used a linear model. The interpretation of the ALE plot is that, given a feature value, the ALE value corresponding to that feature value is the difference to the mean effect of that feature. Put differently, the ALE value is the relative feature effect on the prediction at that feature value.

Effect of the median income

Let's look at the ALE plot for the feature `MedInc` (median income) in more detail:

```
[16]: plot_ale(lr_exp, features=['MedInc']);
```



The ALE on the y-axes of the plot above is in the units of the prediction variable which, in this case, is the value of the house in \$100,000's.

The median income here is in units of \$10,000's.

The main interpretation of the ALE plot is qualitative—fixing the feature value and looking at the ALE plot as a function at that point, the tangent at that point (or the slope of linear interpolation between the closest bin endpoints) shows how sensitive the target prediction is with respect to small changes of the feature value. Since we have a linear regression model, the tangent/slope is the same across the whole feature range so the feature sensitivity is identical at any point in the feature range. We can calculate it by taking the slope of the linear interpolation between any two points of the ALE plot:

```
[17]: slopes = np.array([(v[-1]-v[0])/(f[-1]-f[0])).item() for v, f in zip(lr_exp.ale_values,
    ↪ lr_exp.feature_values)])
slope_med_inc = slopes[feature_names.index('MedInc')]
print(slope_med_inc)
```

```
0.4476000685169465
```

This value can be interpreted that, at any point in the feature range of `MedInc`, the effect of changing it by some amount δ will result in a change in the prediction by approximately $0.4476 \cdot \delta$. In other words, at any point an additional median income of \$10,000 would result in an uplift of the predicted house value by ~\$44,760 dollars.

Note

This interpretation doesn't mean that for any single datapoint the effect will be the same uplift. Rather, the effect is true *on average* for datapoints close to the feature value of interest, i.e. in a bin of size δ .

We can also say a few more quantitative things about this plot. The ALE value for the point `MedInc=6` (\$60,000) is ~1 which has the interpretation that for areas with this median income the model predicts an up-lift of ~\$100,000 *with*

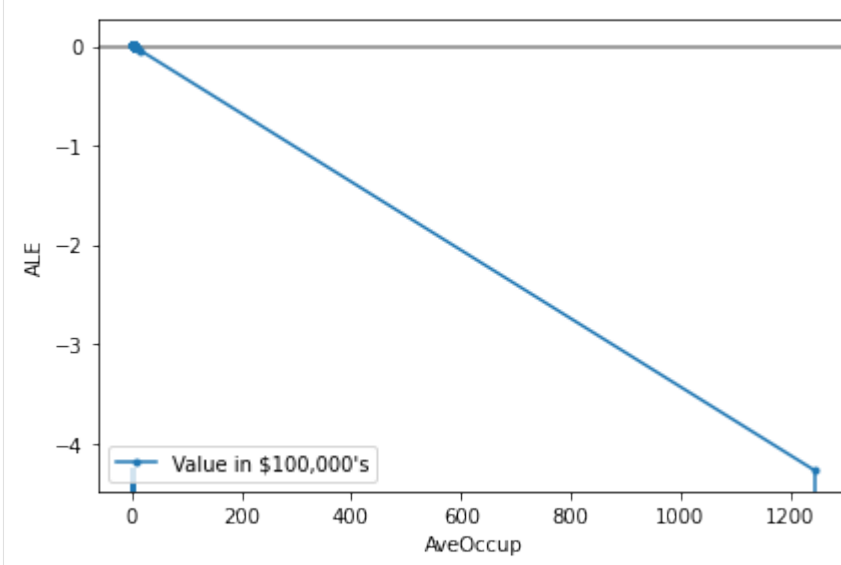
respect to the average effect of `MedInc`. This is because the ALE plots are centered such that the average effect of the feature across the whole range of it is zero.

On the other hand, for neighbourhoods with `MedInc=4` (\$40,000), the ALE value is ~ 0 which indicates that the effect of the feature at this point is the same as the average effect of the feature. For even lower values of `MedInc`, below \$40,000, the feature effect becomes less than the average effect, i.e. a smaller median income in the area brings the predicted house value down with respect to the average feature effect.

Effect of the crime level

An additional feature of the ALE plot is that it shows feature deciles on the x-axis. This helps understand in which regions there is low data density so the ALE plot is interpolating. For example, for the `AveOccup` feature (average number of household members), there appears to be an outlier in the data at over $\sim 1,200$ which causes the plot to linearly interpolate over a large range where there is no data. Note that this can also be seen by the lack of markers on the plot within that large range.

```
[18]: plot_ale(lr_exp, features=['AveOccup']);
```



For linear models this is not an issue as we know the effect is linear across the whole range of the feature, however for non-linear models linear interpolation in feature areas with no data could be unreliable. This is why the use of deciles can help assess in which areas of the feature space the estimated feature effects are more reliable.

Linearity of ALE

It is no surprise that the ALE plots for the linear regression model are linear themselves—the feature effects are after all linear by definition. In fact, the slopes of the ALE lines are exactly the coefficients of the linear regression:

```
[19]: lr.coef_
```

```
[19]: array([ 4.47600069e-01,  9.56752596e-03, -1.24755956e-01,  7.94471254e-01,
        -1.43902596e-06, -3.44307993e-03, -4.18555257e-01, -4.33405135e-01])
```

```
[20]: slopes
```

```
[20]: array([ 4.47600069e-01,  9.56752596e-03, -1.24755956e-01,  7.94471254e-01,
          -1.43902596e-06, -3.44307993e-03, -4.18555257e-01, -4.33405135e-01])
```

```
[21]: np.allclose(lr.coef_, slopes)
```

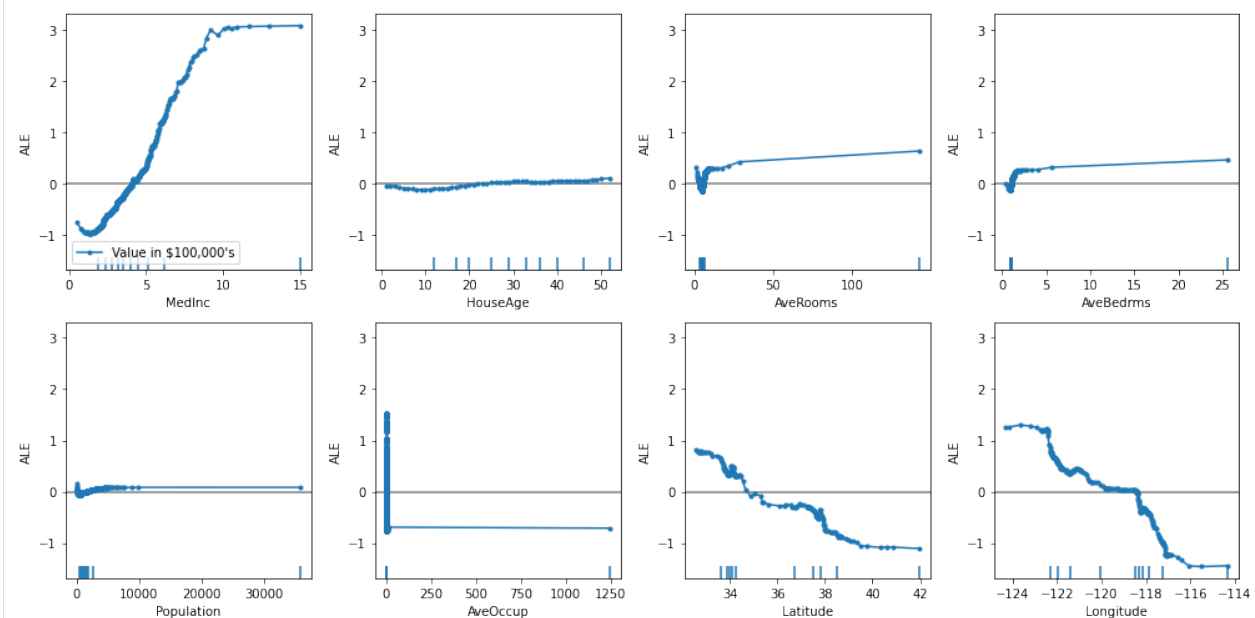
```
[21]: True
```

Thus the slopes of the ALE plots for linear regression have exactly the same interpretation as the coefficients of the learnt model—global feature effects. In fact, we can calculate the ALE effect of linear regression analytically to show that the effect of feature x_i is $\text{ALE}(x_i) = w_i x_i - w_i \mathbb{E}(x_i)$ which is the familiar effect $w_i x_i$ relative to the mean effect of the feature.

ALE for the random forest model

Now let's look at the ALE plots for the non-linear random forest model:

```
[22]: axes = plot_ale(rf_exp, n_cols=4, fig_kw={'figwidth':14, 'figheight': 7});
```



Because the model is no longer linear, the ALE plots are non-linear also and in some cases also non-monotonic. The interpretation of the plots is still the same—the ALE value at a point is the relative feature effect with respect to the mean feature effect, however the non-linear model used shows that the feature effects differ both in shape and magnitude when compared to the linear model.

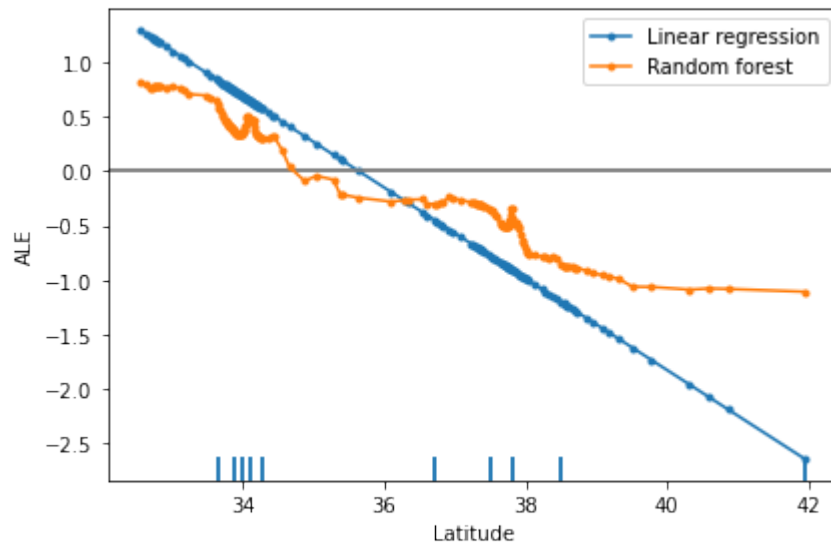
From these plots, it seems that the feature MedInc (median income) has the biggest impact on the prediction. Checking the built-in feature importances of the random forest classifier confirms this:

```
[23]: feature_names[rf[1].feature_importances_.argmax()]
```

```
[23]: 'MedInc'
```

Let's explore the feature Latitude and how its effects are different between the two models. To do this, we can pass in matplotlib axes objects for the `plot_ale` function to plot on:

```
[24]: fig, ax = plt.subplots()
      plot_ale(lr_exp, features=['Latitude'], ax=ax, line_kw={'label': 'Linear regression'});
      plot_ale(rf_exp, features=['Latitude'], ax=ax, line_kw={'label': 'Random forest'});
```



From this plot we can gain a couple of interesting insights:

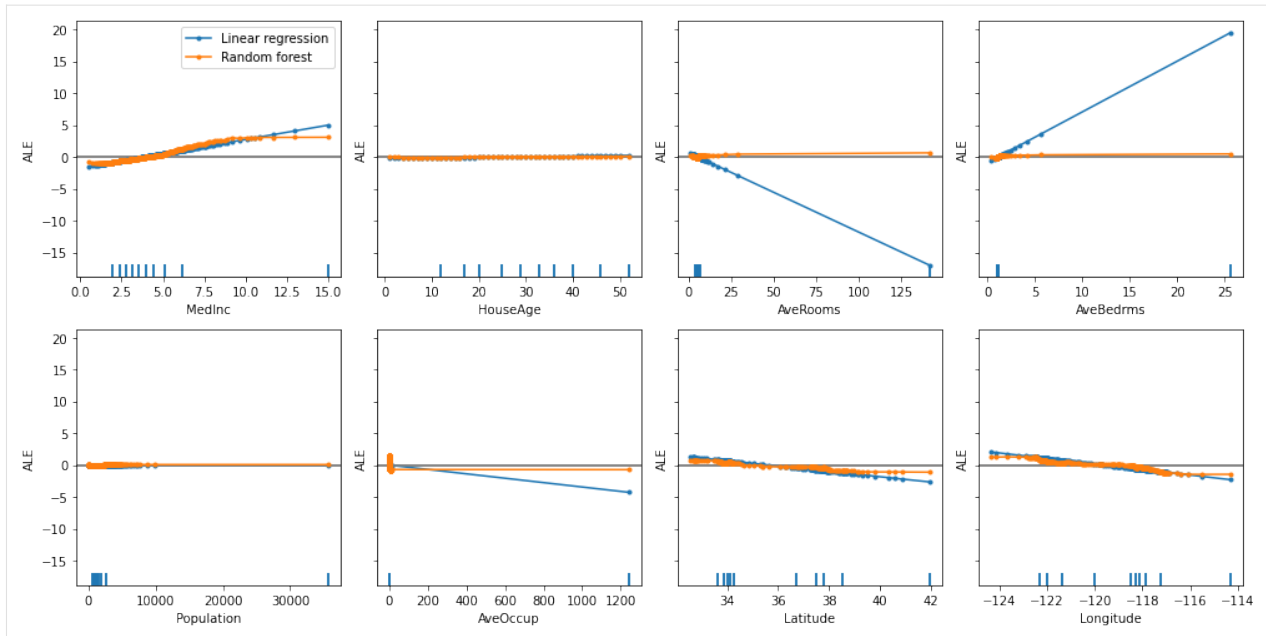
- Whilst for linear regression the feature effects are the same across the range of the values of Latitude, for a random forest model, because it's a more capable predictor, there are intervals of Latitude where different behaviour is observed
- For both models the feature effects of Latitude are negatively correlated, i.e. for areas more South (lower latitude) the effects on house price predictions are positive. However, for the random forest, the ALE curve is sometimes piece-wise constant which tells us that there are regions where the effects are roughly the same for different latitudes (e.g. between latitudes 36 and 37)
- In general, the ALE for a non-linear model doesn't have to be monotonic, although in this case there are only very small departures from monotonicity which may be due to artifacts from the grid-size used to calculate the ALE. It may be useful to experiment with different resolutions of the grid size

Comparing the ALE plots of multiple models on the same axis should be done with care. In general, we can only make qualitative comparisons of the plots between different intervals of the feature values as we have done here.

To compare multiple models and multiple features we can plot the ALE's on a common axis that is big enough to accommodate all features of interest:

```
[26]: fig, ax = plt.subplots(2, 4, sharey='all');

      plot_ale(lr_exp, ax=ax, fig_kw={'figwidth':14, 'figheight': 7},
              line_kw={'label': 'Linear regression'});
      plot_ale(rf_exp, ax=ax, line_kw={'label': 'Random forest'});
```



8.3 Anchors

8.3.1 Anchor explanations for fashion MNIST

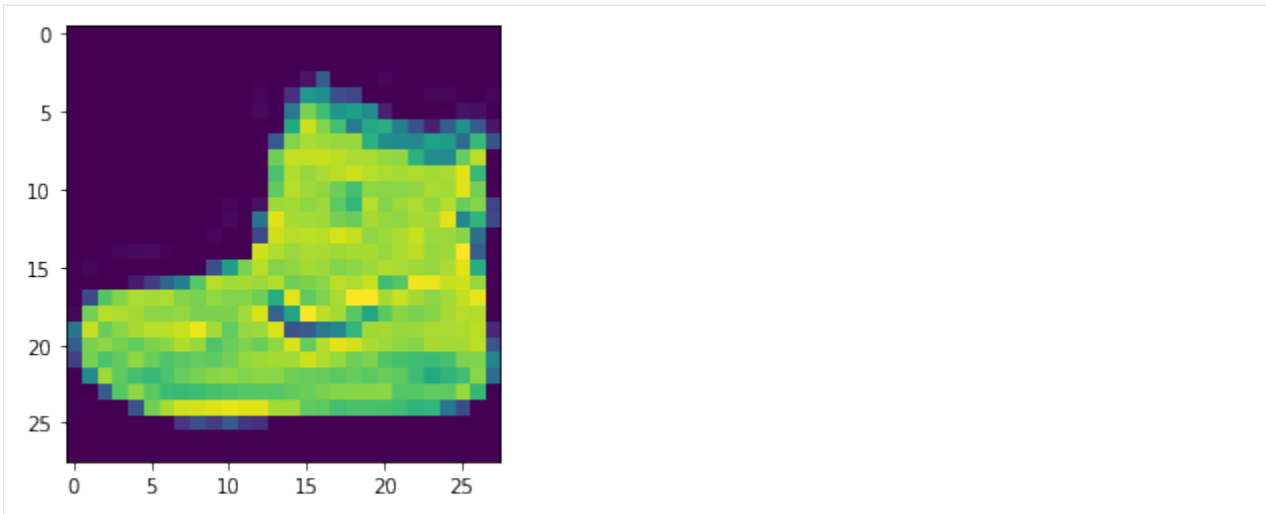
```
[1]: import matplotlib
      %matplotlib inline
      import matplotlib.pyplot as plt
      import numpy as np
      import tensorflow as tf
      from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input
      from tensorflow.keras.models import Model
      from tensorflow.keras.utils import to_categorical
      from alibi.explainers import AnchorImage
```

Load and prepare fashion MNIST data

```
[2]: (x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
      print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)

x_train shape: (60000, 28, 28) y_train shape: (60000,)
```

```
[3]: idx = 0
      plt.imshow(x_train[idx]);
```



Scale, reshape and categorize data

```
[4]: x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = np.reshape(x_train, x_train.shape + (1,))
x_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)
```

x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
y_train shape: (60000, 10) y_test shape: (10000, 10)

Define CNN model

```
[5]: def model():
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(filters=64, kernel_size=2, padding='same', activation='relu')(x_in)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)

    x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)

    x = Flatten()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.5)(x)
    x_out = Dense(10, activation='softmax')(x)

    cnn = Model(inputs=x_in, outputs=x_out)
    cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    return cnn
```



```
[6]: cnn = model()
      cnn.summary()
```

```
Model: "model"
```

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d (Conv2D)	(None, 28, 28, 64)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 14, 14, 32)	8224
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 32)	0
dropout_1 (Dropout)	(None, 7, 7, 32)	0
flatten (Flatten)	(None, 1568)	0
dense (Dense)	(None, 256)	401664
dropout_2 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2570
=====		
Total params: 412,778		
Trainable params: 412,778		
Non-trainable params: 0		
=====		

Train model

```
[7]: cnn.fit(x_train, y_train, batch_size=64, epochs=3)
```

```
Train on 60000 samples
```

```
Epoch 1/3
```

```
60000/60000 [=====] - 29s 481us/sample - loss: 0.5932 - acc: 0.7819
```

```
Epoch 2/3
```

```
60000/60000 [=====] - 33s 542us/sample - loss: 0.4066 - acc: 0.8506
```

```
Epoch 3/3
```

```
60000/60000 [=====] - 32s 525us/sample - loss: 0.3624 - acc: 0.8681
```

```
[7]: <tensorflow.python.keras.callbacks.History at 0x7fae6dd5cb70>
```

```
[8]: # Evaluate the model on test set
score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])
```

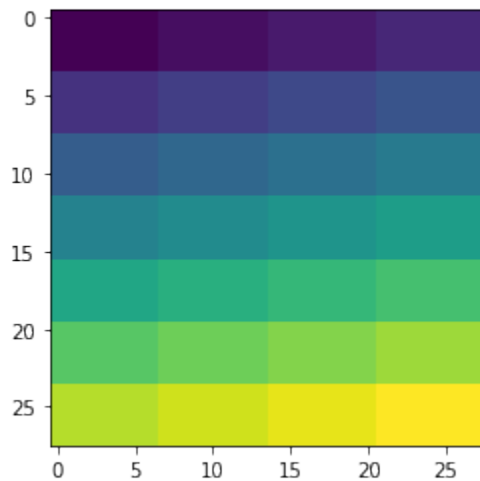
```
Test accuracy:  0.8867
```

Define superpixels

Function to generate rectangular superpixels for a given image. Alternatively, use one of the built in methods. It is important to have meaningful superpixels in order to generate a useful explanation. Please check scikit-image's [segmentation methods](#) (*felzenszwalb*, *slic* and *quickshift* built in the explainer) for more information on the built in methods.

```
[9]: def superpixel(image, size=(4, 7)):
    segments = np.zeros([image.shape[0], image.shape[1]])
    row_idx, col_idx = np.where(segments == 0)
    for i, j in zip(row_idx, col_idx):
        segments[i, j] = int((image.shape[1]/size[1]) * (i//size[0]) + j//size[1])
    return segments
```

```
[10]: segments = superpixel(x_train[idx])
plt.imshow(segments);
```



Define prediction function

```
[11]: predict_fn = lambda x: cnn.predict(x)
```

Initialize anchor image explainer

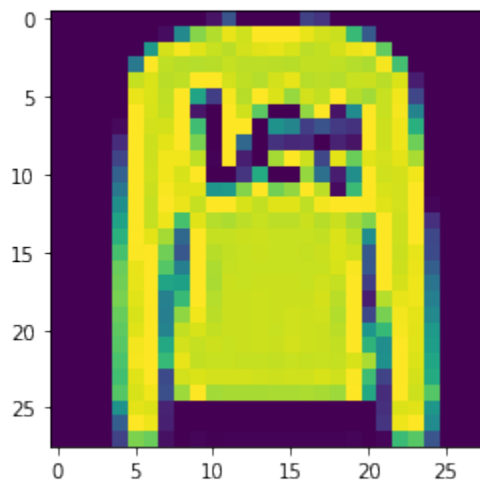
```
[12]: image_shape = x_train[idx].shape
explainer = AnchorImage(predict_fn, image_shape, segmentation_fn=superpixel)
```

Explain a prediction

The explanation returns a mask with the superpixels that constitute the anchor.

Image to be explained:

```
[13]: i = 1
image = x_test[i]
plt.imshow(image[:, :, 0]);
```



Model prediction:

```
[14]: cnn.predict(image.reshape(1, 28, 28, 1)).argmax()
```

```
[14]: 2
```

The predicted category correctly corresponds to the class Pullover:

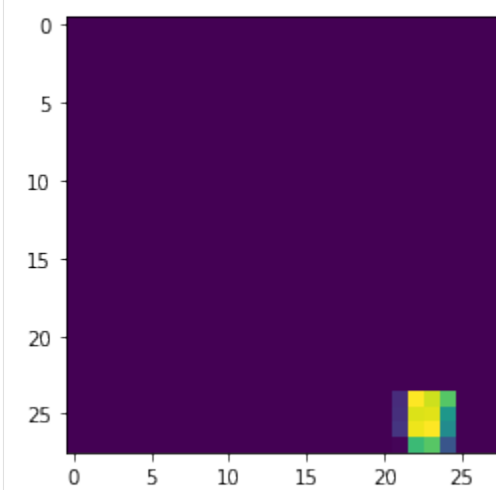
Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Generate explanation:

```
[15]: explanation = explainer.explain(image, threshold=.95, p_sample=.8, seed=0)
```

Show anchor:

```
[16]: plt.imshow(explanation.anchor[:, :, 0]);
```



From the example, it looks like the end of the sleeve alone is sufficient to predict a pullover.

8.3.2 Anchor explanations for ImageNet

```
[1]: import tensorflow as tf
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.applications.inception_v3 import InceptionV3, preprocess_input,
↳ decode_predictions
from alibi.datasets import load_cats
from alibi.explainers import AnchorImage
```

Load InceptionV3 model pre-trained on ImageNet

```
[2]: model = InceptionV3(weights='imagenet')
```

Load and pre-process sample images

The `load_cats` function loads a small sample of images of various cat breeds.

```
[3]: image_shape = (299, 299, 3)
data, labels = load_cats(target_size=image_shape[:2], return_X_y=True)
print(f'Images shape: {data.shape}')
```

```
Images shape: (4, 299, 299, 3)
```

Apply image preprocessing, make predictions and map predictions back to categories. The output label is a tuple which consists of the class name, description and the prediction probability.

```
[4]: images = preprocess_input(data)
preds = model.predict(images)
label = decode_predictions(preds, top=3)
print(label[0])

1/1 [=====] - 4s 4s/step
[('n02123045', 'tabby', 0.82086897), ('n02123159', 'tiger_cat', 0.14372891), ('n02124075', 'Egyptian_cat', 0.01642174)]
```

Define prediction function

```
[5]: predict_fn = lambda x: model.predict(x)
```

Initialize anchor image explainer

The segmentation function will be used to generate superpixels. It is important to have meaningful superpixels in order to generate a useful explanation. Please check scikit-image's [segmentation methods](#) (*felzenszwalb*, *slic* and *quickshift* built in the explainer) for more information.

In the example, the pixels not in the proposed anchor will take the average value of their superpixel. Another option is to superimpose the pixel values from other images which can be passed as a numpy array to the `images_background` argument.

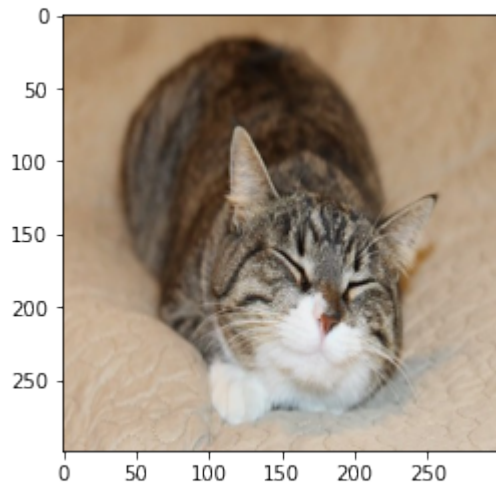
```
[6]: segmentation_fn = 'slic'
kwargs = {'n_segments': 15, 'compactness': 20, 'sigma': .5, 'start_label': 0}
explainer = AnchorImage(predict_fn, image_shape, segmentation_fn=segmentation_fn,
                        segmentation_kwargs=kwargs, images_background=None)

1/1 [=====] - 2s 2s/step
```

Explain a prediction

The explanation of the below image returns a mask with the superpixels that constitute the anchor.

```
[7]: i = 0
plt.imshow(data[i]);
```



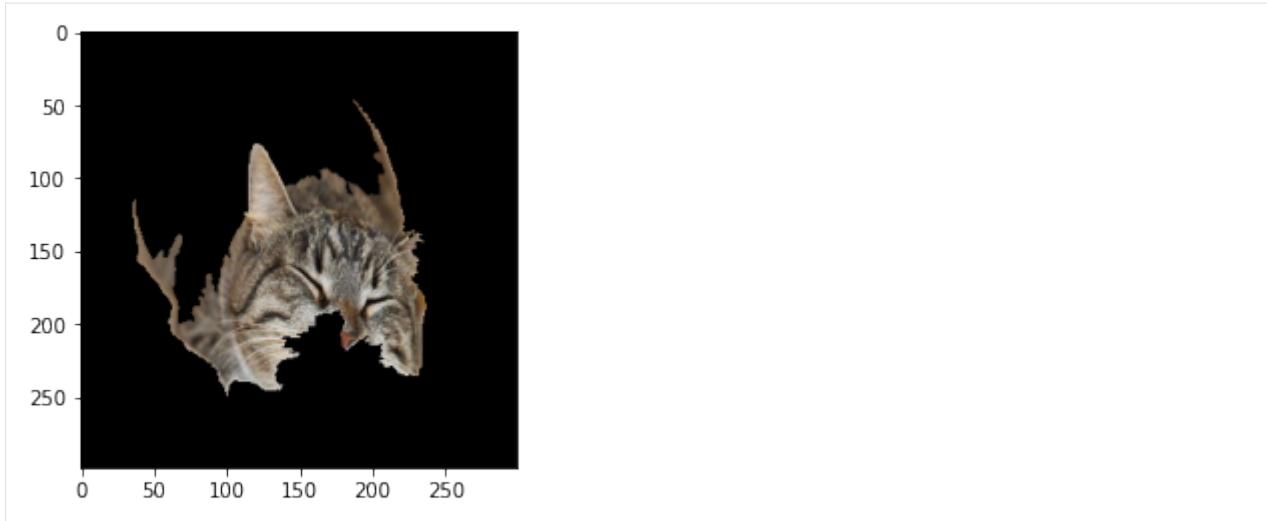
The *threshold*, *p_sample* and *tau* parameters are also key to generate a sensible explanation and ensure fast enough convergence. The *threshold* defines the minimum fraction of samples for a candidate anchor that need to lead to the same prediction as the original instance. While a higher threshold gives more confidence in the anchor, it also leads to longer computation time. *p_sample* determines the fraction of superpixels that are changed to either the average value of the superpixel or the pixel value for the superimposed image. The pixels in the proposed anchors are of course unchanged. The parameter *tau* determines when we assume convergence. A bigger value for *tau* means faster convergence but also looser anchor restrictions.

```
[8]: image = images[i]
np.random.seed(0)
explanation = explainer.explain(image, threshold=.95, p_sample=.5, tau=0.25)

1/1 [=====] - 0s 25ms/step
4/4 [=====] - 5s 304ms/step
4/4 [=====] - 1s 437ms/step
4/4 [=====] - 1s 448ms/step
4/4 [=====] - 1s 436ms/step
```

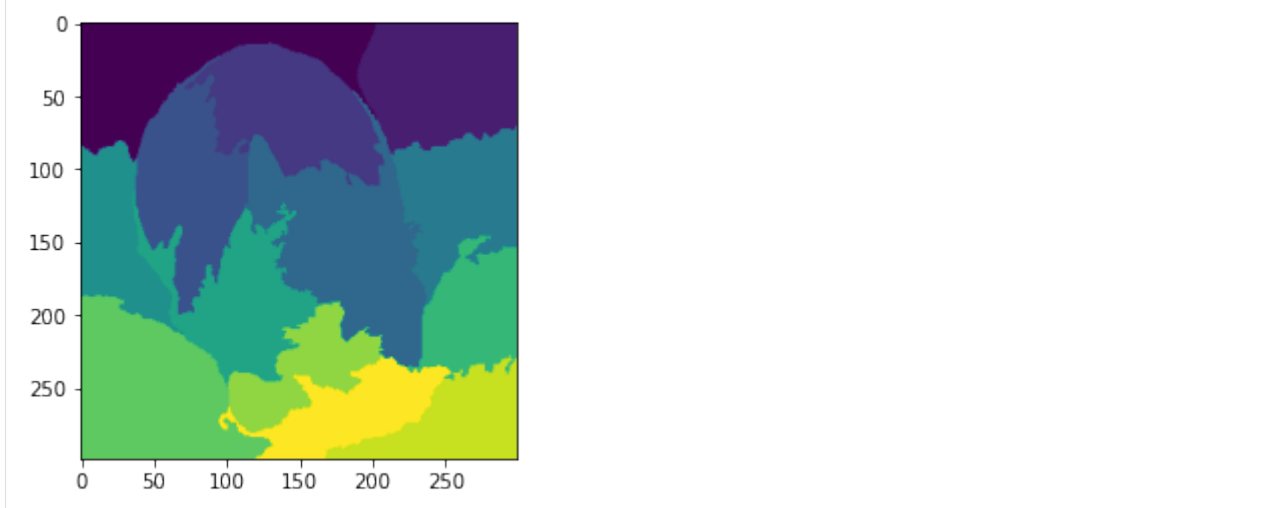
Superpixels in the anchor:

```
[9]: plt.imshow(explanation.anchor);
```



A visualization of all the superpixels:

```
[10]: plt.imshow(explanation.segments);
```



8.3.3 Anchor explanations for income prediction

In this example, we will explain predictions of a Random Forest classifier whether a person will make more or less than \$50k based on characteristics like age, marital status, gender or occupation. The features are a mixture of ordinal and categorical data and will be pre-processed accordingly.

```
[1]: import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from alibi.explainers import AnchorTabular
from alibi.datasets import fetch_adult
```

Load adult dataset

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of categorical variables to numbers which are required for formatting the output of the Anchor explainer.

```
[2]: adult = fetch_adult()
      adult.keys()

[2]: dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])

[3]: data = adult.data
      target = adult.target
      feature_names = adult.feature_names
      category_map = adult.category_map
```

Note that for your own datasets you can use our utility function `gen_category_map` to create the category map:

```
[4]: from alibi.utils import gen_category_map
```

Define shuffled training and test set

```
[5]: np.random.seed(0)
      data_perm = np.random.permutation(np.c_[data, target])
      data = data_perm[:, :-1]
      target = data_perm[:, -1]

[6]: idx = 30000
      X_train, Y_train = data[:idx, :], target[:idx]
      X_test, Y_test = data[idx+1:, :], target[idx+1:]
```

Create feature transformation pipeline

Create feature pre-processor. Needs to have ‘fit’ and ‘transform’ methods. Different types of pre-processing can be applied to all or part of the features. In the example below we will standardize ordinal features and apply one-hot-encoding to categorical features.

Ordinal features:

```
[7]: ordinal_features = [x for x in range(len(feature_names)) if x not in list(category_map.
    ↪ keys())]
      ordinal_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
    ↪ ('scaler', StandardScaler())])
```

Categorical features:

```
[8]: categorical_features = list(category_map.keys())
      categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
    ↪ ('onehot', OneHotEncoder(handle_unknown='ignore'
    ↪ ')))])
```

Combine and fit:


```

[9]: preprocessor = ColumnTransformer(transformers=[('num', ordinal_transformer, ordinal_
↳ features),

                                                    ('cat', categorical_transformer,
↳ categorical_features)])
preprocessor.fit(X_train)

[9]: ColumnTransformer(n_jobs=None, remainder='drop', sparse_threshold=0.3,
                        transformer_weights=None,
                        transformers=[('num',
                                     Pipeline(memory=None,
                                             steps=[('imputer',
                                                    SimpleImputer(add_indicator=False,
                                                                    copy=True,
                                                                    fill_value=None,
                                                                    missing_values=nan,
                                                                    strategy='median',
                                                                    verbose=0)),
                                                    ('scaler',
                                                       StandardScaler(copy=True,
                                                                    with_mean=True,
                                                                    with_std=True))],
                                             verbose=False),
                                     [0, 8, 9, 10]),
                                    ('cat',
                                     Pipeline(memory=None,
                                             steps=[('imputer',
                                                    SimpleImputer(add_indicator=False,
                                                                    copy=True,
                                                                    fill_value=None,
                                                                    missing_values=nan,
                                                                    strategy='median',
                                                                    verbose=0)),
                                                    ('onehot',
                                                       OneHotEncoder(categories='auto',
                                                                    drop=None,
                                                                    dtype=<class 'numpy.
↳ float64'>,
                                                                    handle_unknown='ignore',
                                                                    sparse=True))],
                                             verbose=False),
                                     [1, 2, 3, 4, 5, 6, 7, 11])],
                        verbose=False)

```

Train Random Forest model

Fit on pre-processed (imputing, OHE, standardizing) data.

```
[10]: np.random.seed(0)
      clf = RandomForestClassifier(n_estimators=50)
      clf.fit(preprocessor.transform(X_train), Y_train)

[10]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                             criterion='gini', max_depth=None, max_features='auto',
                             max_leaf_nodes=None, max_samples=None,
                             min_impurity_decrease=0.0, min_impurity_split=None,
                             min_samples_leaf=1, min_samples_split=2,
                             min_weight_fraction_leaf=0.0, n_estimators=50,
                             n_jobs=None, oob_score=False, random_state=None,
                             verbose=0, warm_start=False)
```

Define predict function

```
[11]: predict_fn = lambda x: clf.predict(preprocessor.transform(x))
      print('Train accuracy: ', accuracy_score(Y_train, predict_fn(X_train)))
      print('Test accuracy: ', accuracy_score(Y_test, predict_fn(X_test)))

Train accuracy:  0.9655333333333334
Test accuracy:  0.855859375
```

Initialize and fit anchor explainer for tabular data

```
[12]: explainer = AnchorTabular(predict_fn, feature_names, categorical_names=category_map,
      ↪seed=1)
```

Discretize the ordinal features into quartiles

```
[13]: explainer.fit(X_train, disc_perc=[25, 50, 75])

[13]: AnchorTabular(meta={
      'name': 'AnchorTabular',
      'type': ['blackbox'],
      'explanations': ['local'],
      'params': {'seed': 1, 'disc_perc': [25, 50, 75]}
    })
```

Getting an anchor

Below, we get an anchor for the prediction of the first observation in the test set. An anchor is a sufficient condition - that is, when the anchor holds, the prediction should be the same as the prediction for this instance.

```
[14]: idx = 0
      class_names = adult.target_names
      print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])

Prediction:  <=50K
```

We set the precision threshold to 0.95. This means that predictions on observations where the anchor holds will be the same as the prediction on the explained instance at least 95% of the time.

```
[15]: explanation = explainer.explain(X_test[idx], threshold=0.95)
print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('Coverage: %.2f' % explanation.coverage)
```

```
Anchor: Marital Status = Separated AND Sex = Female
Precision: 0.95
Coverage: 0.18
```

...or not?

Let's try getting an anchor for a different observation in the test set - one for the which the prediction is >50K.

```
[16]: idx = 6
class_names = adult.target_names
print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])

explanation = explainer.explain(X_test[idx], threshold=0.95)
print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('Coverage: %.2f' % explanation.coverage)
```

```
Prediction: >50K
```

```
Could not find an result satisfying the 0.95 precision constraint. Now returning the
↳ best non-eligible result.
```

```
Anchor: Capital Loss > 0.00 AND Relationship = Husband AND Marital Status = Married AND
↳ Age > 37.00 AND Race = White AND Country = United-States AND Sex = Male
Precision: 0.71
Coverage: 0.05
```

Notice how no anchor is found!

This is due to the imbalanced dataset (roughly 25:75 high:low earner proportion), so during the sampling stage feature ranges corresponding to low-earners will be oversampled. This is a feature because it can point out an imbalanced dataset, but it can also be fixed by producing balanced datasets to enable anchors to be found for either class.

8.3.4 Anchor explanations on the Iris dataset

```
[1]: import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from alibi.explainers import AnchorTabular
```

Load iris dataset

```
[2]: dataset = load_iris()
feature_names = dataset.feature_names
class_names = list(dataset.target_names)
```

Define training and test set

```
[3]: idx = 145
X_train, Y_train = dataset.data[:idx, :], dataset.target[:idx]
X_test, Y_test = dataset.data[idx+1:, :], dataset.target[idx+1:]
```

Train Random Forest model

```
[4]: np.random.seed(0)
clf = RandomForestClassifier(n_estimators=50)
clf.fit(X_train, Y_train)

[4]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                             criterion='gini', max_depth=None, max_features='auto',
                             max_leaf_nodes=None, max_samples=None,
                             min_impurity_decrease=0.0, min_impurity_split=None,
                             min_samples_leaf=1, min_samples_split=2,
                             min_weight_fraction_leaf=0.0, n_estimators=50,
                             n_jobs=None, oob_score=False, random_state=None,
                             verbose=0, warm_start=False)
```

Define predict function

```
[5]: predict_fn = lambda x: clf.predict_proba(x)
```

Initialize and fit anchor explainer for tabular data

```
[6]: explainer = AnchorTabular(predict_fn, feature_names)
```

Discretize the ordinal features into quartiles

```
[7]: explainer.fit(X_train, disc_perc=(25, 50, 75))

[7]: AnchorTabular(meta={
    'name': 'AnchorTabular',
    'type': ['blackbox'],
    'explanations': ['local'],
    'params': {'seed': None, 'disc_perc': (25, 50, 75)}
})
```

Getting an anchor

Below, we get an anchor for the prediction of the first observation in the test set. An anchor is a sufficient condition - that is, when the anchor holds, the prediction should be the same as the prediction for this instance.

```
[8]: idx = 0
print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])

Prediction:  virginica
```

We set the precision threshold to 0.95. This means that predictions on observations where the anchor holds will be the same as the prediction on the explained instance at least 95% of the time.

```
[9]: explanation = explainer.explain(X_test[idx], threshold=0.95)
print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('Coverage: %.2f' % explanation.coverage)

Anchor: petal width (cm) > 1.80 AND sepal width (cm) <= 2.80
Precision: 0.98
Coverage: 0.32
```

8.3.5 Anchor explanations for movie sentiment

In this example, we will explain why a certain sentence is classified by a logistic regression as having negative or positive sentiment. The logistic regression is trained on negative and positive movie reviews.

Note

To enable support for the anchor text language models, you may need to run

```
pip install alibi[tensorflow]
```

```
[1]: import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3' # surpressing some transformers' output

import spacy
import string
import numpy as np

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

from alibi.explainers import AnchorText
from alibi.datasets import fetch_movie_sentiment
from alibi.utils import spacy_model
from alibi.utils import DistilbertBaseUncased, BertBaseUncased, RobertaBase
```

Load movie review dataset

The `fetch_movie_sentiment` function returns a Bunch object containing the features, the targets and the target names for the dataset.

```
[2]: movies = fetch_movie_sentiment()
      movies.keys()

[2]: dict_keys(['data', 'target', 'target_names'])
```

```
[3]: data = movies.data
      labels = movies.target
      target_names = movies.target_names
```

Define shuffled training, validation and test set

```
[4]: train, test, train_labels, test_labels = train_test_split(data, labels, test_size=.2,
      ↪ random_state=42)
      train, val, train_labels, val_labels = train_test_split(train, train_labels, test_size=.
      ↪ 1, random_state=42)
      train_labels = np.array(train_labels)
      test_labels = np.array(test_labels)
      val_labels = np.array(val_labels)
```

Apply CountVectorizer to training set

```
[5]: vectorizer = CountVectorizer(min_df=1)
      vectorizer.fit(train)

[5]: CountVectorizer()
```

Fit model

```
[6]: np.random.seed(0)
      clf = LogisticRegression(solver='liblinear')
      clf.fit(vectorizer.transform(train), train_labels)

[6]: LogisticRegression(solver='liblinear')
```

Define prediction function

```
[7]: predict_fn = lambda x: clf.predict(vectorizer.transform(x))
```

Make predictions on train and test sets

```
[8]: preds_train = predict_fn(train)
     preds_val = predict_fn(val)
     preds_test = predict_fn(test)
     print('Train accuracy: %.3f' % accuracy_score(train_labels, preds_train))
     print('Validation accuracy: %.3f' % accuracy_score(val_labels, preds_val))
     print('Test accuracy: %.3f' % accuracy_score(test_labels, preds_test))
```

```
Train accuracy: 0.980
Validation accuracy: 0.754
Test accuracy: 0.759
```

Load spaCy model

English multi-task CNN trained on OntoNotes, with GloVe vectors trained on Common Crawl. Assigns word vectors, context-specific token vectors, POS tags, dependency parse and named entities.

```
[9]: model = 'en_core_web_md'
     spacy_model(model=model)
     nlp = spacy.load(model)
```

Instance to be explained

```
[10]: class_names = movies.target_names

      # select instance to be explained
      text = data[4]
      print("* Text: %s" % text)

      # compute class prediction
      pred = class_names[predict_fn([text])[0]]
      alternative = class_names[1 - predict_fn([text])[0]]
      print("* Prediction: %s" % pred)
```

```
* Text: a visually flashy but narratively opaque and emotionally vapid exercise in style,
↳and mystification .
* Prediction: negative
```

Initialize anchor text explainer with unknown sampling

- `sampling_strategy='unknown'` means we will perturb examples by replacing words with UNKs.

```
[11]: explainer = AnchorText(
      predictor=predict_fn,
      sampling_strategy='unknown',
      nlp=nlp,
    )
```

Explanation

```
[12]: explanation = explainer.explain(text, threshold=0.95)
```

Let us now take a look at the anchor. The word `flashy` basically guarantees a negative prediction.

```
[13]: print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('\nExamples where anchor applies and model predicts %s:' % pred)
print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_true']]))
print('\nExamples where anchor applies and model predicts %s:' % alternative)
print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_false']]))
```

```
Anchor: flashy
Precision: 0.99
```

Examples where anchor applies and model predicts negative:

```
a UNK flashy UNK UNK opaque and emotionally vapid exercise in style UNK mystification .
a UNK flashy UNK UNK UNK and emotionally UNK exercise UNK UNK and UNK UNK
a UNK flashy UNK narratively opaque UNK UNK UNK exercise in style and UNK UNK
UNK visually flashy UNK narratively UNK and emotionally UNK UNK UNK UNK UNK
↳mystification .
UNK UNK flashy UNK UNK opaque and emotionally UNK UNK in UNK and UNK .
a visually flashy but UNK UNK and UNK UNK UNK in style UNK mystification .
a visually flashy but UNK opaque UNK emotionally vapid UNK in UNK and mystification .
a UNK flashy but narratively UNK UNK emotionally vapid exercise in style UNK
↳mystification UNK
a UNK flashy but narratively opaque UNK emotionally vapid exercise in style and
↳mystification .
a visually flashy UNK UNK opaque UNK UNK UNK exercise in UNK UNK UNK .
```

Examples where anchor applies and model predicts positive:

```
UNK UNK flashy but narratively UNK and UNK UNK UNK in style and UNK UNK
```

Initialize anchor text explainer with word similarity sampling

Let's try this with another perturbation distribution, namely one that replaces words by similar words instead of UNKs.

```
[14]: explainer = AnchorText(
    predictor=predict_fn,
    sampling_strategy='similarity', # replace masked words by simialar words
    nlp=nlp,                       # spacy object
    sample_proba=0.5,              # probability of a word to be masked and replace
    ↳by as similar word
)
```

```
[15]: explanation = explainer.explain(text, threshold=0.95)
```

The anchor now shows that we need more to guarantee the negative prediction:

```
[16]: print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
```

(continues on next page)

(continued from previous page)

```
print('\nExamples where anchor applies and model predicts %s:' % pred)
print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_true']]))
print('\nExamples where anchor applies and model predicts %s:' % alternative)
print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_false']]))
```

Anchor: exercise AND vapid

Precision: 0.99

Examples where anchor applies and model predicts negative:

that visually flashy but tragically opaque and emotionally vapid exercise under genre.
 ↪ and glorification .
 another provably flashy but hysterically bulky and emotionally vapid exercise arounds.
 ↪ style and authorization .
 that- visually flashy but narratively opaque and politically vapid exercise in style and.
 ↪ mystification .
 a unintentionally decal but narratively thick and emotionally vapid exercise in.
 ↪ unflattering and mystification .
 the purposely flashy but narratively rosy and emotionally vapid exercise in style and.
 ↪ mystification .
 thievery intentionally flashy but hysterically gray and anally vapid exercise in style.
 ↪ and mystification .
 a irrationally flashy but narratively smoothness and purposefully vapid exercise near.
 ↪ style and diction .
 a medio flashy but narratively blue and economically vapid exercise since style and.
 ↪ intuition .
 a visually flashy but narratively opaque and anally vapid exercise onwards style and.
 ↪ mystification .
 each purposefully flashy but narratively gorgeous and emotionally vapid exercise in.
 ↪ style and mystification .

Examples where anchor applies and model predicts positive:

a visually punchy but tragically opaque and hysterically vapid exercise in minimalist.
 ↪ and mystification .
 a visually discernible but realistically posh and physically vapid exercise around style.
 ↪ and determination .

We can make the token perturbation distribution sample words that are more similar to the ground truth word via the `top_n` argument. Smaller values (default=100) should result in sentences that are more coherent and thus more in the distribution of natural language which could influence the returned anchor. By setting the `use_proba` to `True`, the sampling distribution for perturbed tokens is proportional to the similarity score between the possible perturbations and the original word. We can also put more weight on similar words via the `temperature` argument. Lower values of `temperature` increase the sampling weight of more similar words. The following example will perturb tokens in the original sentence with probability equal to `sample_proba`. The sampling distribution for the perturbed tokens is proportional to the similarity score between the ground truth word and each of the `top_n` words.

```
[17]: explainer = AnchorText(
    predictor=predict_fn,
    sampling_strategy='similarity', # replace masked words by simialar words
    nlp=nlp,                       # spacy object
    use_proba=True,                # sample according to the similiary distribution
    sample_proba=0.5,              # probability of a word to be masked and replace by.
    ↪ as similar word
    top_n=20,                      # consider only top 20 words most similar words
```

(continues on next page)

(continued from previous page)

```

    temperature=0.2                # higher temperature implies more randomness when
    ↪ sampling
)

```

```
[18]: explanation = explainer.explain(text, threshold=0.95)
```

```
[19]: print('Anchor: %s' % (' AND '.join(explanation.anchor)))
      print('Precision: %.2f' % explanation.precision)
      print('\nExamples where anchor applies and model predicts %s:' % pred)
      print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_true']]))
      print('\nExamples where anchor applies and model predicts %s:' % alternative)
      print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_false']]))

```

```

Anchor: exercise AND flashy
Precision: 1.00

```

Examples where anchor applies and model predicts negative:

```

a visually flashy but sarcastically brown and reflexively vapid exercise between style
↪ and mystification .
this visually flashy but intentionally shiny and emotionally vapid exercise in style and
↪ appropriation .
a visually flashy but narratively glossy and critically vapid exercise in accentuate and
↪ omission .
a visually flashy but historically glossy and purposely rapid exercise within stylesheet
↪ and equivocation .
each visually flashy but intently opaque and emotionally quickie exercise throughout
↪ style and mystification .
that reflexively flashy but narratively opaque and romantically melodramatic exercise
↪ within style and mystification .
a equally flashy but narratively boxy and emotionally predictable exercise in classism
↪ and exaggeration .
a visually flashy but narratively opaque and emotionally vapid exercise between style
↪ and mystification .
a visually flashy but emphatically opaque and emotionally vapid exercise walkthrough
↪ classism and mystification .
a verbally flashy but sarcastically opaque and emotionally dramatic exercise in design
↪ and mystification .

```

Examples where anchor applies and model predicts positive:

```

that visually flashy but narratively boxy and reflexively insignificant exercise outside
↪ minimalist and appropriation .

```

Initialize language model

Because the Language Model is computationally demanding, we can run it on the GPU. Note that this is optional, and we can run the explainer on a non-GPU machine too.

```
[20]: # the code runs for non-GPU machines too
os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"]="0"
```

We provide support for three transformer-based language models: `DistilbertBaseUncased`, `BertBaseUncased`, and `RobertaBase`. We initialize the language model as follows:

```
[21]: # language_model = RobertaBase()
# language_model = BertBaseUncased()
language_model = DistilbertBaseUncased()
```

Some layers from the model checkpoint at `distilbert-base-uncased` were not used when initializing `TFDistilBertForMaskedLM`: `['activation_13']`

- This IS expected if you are initializing `TFDistilBertForMaskedLM` from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a `BertForSequenceClassification` model from a `BertForPreTraining` model).
- This IS NOT expected if you are initializing `TFDistilBertForMaskedLM` from the checkpoint of a model that you expect to be exactly identical (initializing a `BertForSequenceClassification` model from a `BertForSequenceClassification` model).

All the layers of `TFDistilBertForMaskedLM` were initialized from the model checkpoint at `distilbert-base-uncased`.

If your task is similar to the task the model of the checkpoint was trained on, you can already use `TFDistilBertForMaskedLM` for predictions without further training.

Initialize anchor text explainer with language_model sampling (parallel filling)

- `sampling_strategy='language_model'` means that the words will be sampled according to the output distribution predicted by the language model
- `filling='parallel'` means the only one forward pass is performed. The words are the sampled independently of one another.

```
[22]: # initialize explainer
explainer = AnchorText(
    predictor=predict_fn,
    sampling_strategy="language_model", # use language model to predict the masked
    words
    language_model=language_model, # language model to be used
    filling="parallel", # just one pass through the transformer
    sample_proba=0.5, # probability of masking a word
    frac_mask_templates=0.1, # fraction of masking templates (smaller value
    -> faster, less diverse)
    use_proba=True, # use words distribution when sampling (if
    False sample uniform)
    top_n=20, # consider the fist 20 most likely words
    temperature=1.0, # higher temperature implies more randomness
    when sampling
    stopwords=['and', 'a', 'but', 'in'], # those words will not be sampled
```

(continues on next page)

(continued from previous page)

```

    batch_size_lm=32,                                # language model maximum batch size
)

```

```
[23]: explanation = explainer.explain(text, threshold=0.95)
```

```
[24]: print('Anchor: %s' % (' AND '.join(explanation.anchor)))
      print('Precision: %.2f' % explanation.precision)
      print('\nExamples where anchor applies and model predicts %s:' % pred)
      print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_true']]))
      print('\nExamples where anchor applies and model predicts %s:' % alternative)
      print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_false']]))

```

```

Anchor: exercise AND flashy AND opaque
Precision: 0.95

```

Examples where anchor applies and model predicts negative:

```

a visually flashy but visually opaque and politically photographic exercise in style and
↳mystification.
a visually flashy but visually opaque and emotionally expressive exercise in style and
↳mystification.
a visually flashy but visually opaque and socially visual exercise in style and
↳mystification.
a visually flashy but ultimately opaque and visually dramatic exercise in style and
↳mystification.
a visually flashy but often opaque and highly conscious exercise in style and
↳mystification.
a visually flashy but historically opaque and intensely thorough exercise in style and
↳mystification.
a visually flashy but socially opaque and visually an exercise in style and
↳mystification.
a visually flashy but emotionally opaque and socially an exercise in style and
↳mystification.
a visually flashy but emotionally opaque and visually creative exercise in style and
↳mystification.
a visually flashy but sometimes opaque and subtly photographic exercise in style and
↳mystification.

```

Examples where anchor applies and model predicts positive:

```

a visually flashy but visually opaque and deeply enjoyable exercise in imagination and
↳mystification.
a visually flashy but somewhat opaque and highly an exercise in reflection and
↳mystification.
a visually flashy but narratively opaque and deeply imaginative exercise in style and
↳mystification.
a visually flashy but narratively opaque and visually challenging exercise in style and
↳mystification.
a visually flashy but narratively opaque and intensely challenging exercise in style and
↳mystification.
a surprisingly flashy but narratively opaque and highly rigorous exercise in style and
↳mystification.
a very flashy but narratively opaque and highly imaginative exercise in style and
↳mystification.

```

Initialize anchor text explainer with language_model sampling (autoregressive filling)

- filling='autoregressive' means that the words are sampled one at the time (autoregressive). Thus, following words to be predicted will be conditioned on the previously generated words.
- frac_mask_templates=1 in this mode (overwriting it with any other value will not be considered).
- **This procedure is computationally expensive.**

```
[25]: # initialize explainer
explainer = AnchorText(
    predictor=predict_fn,
    sampling_strategy="language_model", # use language model to predict the masked words
    language_model=language_model,    # language model to be used
    filling="autoregressive",          # just one pass through the transformer
    sample_proba=0.5,                  # probability of masking a word
    use_proba=True,                    # use words distribution when sampling (if
    ↪ False sample uniform)
    top_n=20,                          # consider the first 20 most likely words
    stopwords=['and', 'a', 'but', 'in'] # those words will not be sampled
)
```

```
[26]: explanation = explainer.explain(text, threshold=0.95, batch_size=10, coverage_
    ↪ samples=100)
```

```
[27]: print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('\nExamples where anchor applies and model predicts %s:' % pred)
print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_true']]))
print('\nExamples where anchor applies and model predicts %s:' % alternative)
print('\n'.join([x for x in explanation.raw['examples'][-1]['covered_false']]))
```

```
Anchor: flashy AND exercise AND vapid
Precision: 0.96
```

Examples where anchor applies and model predicts negative:

```
a visually flashy but emotionally opaque and emotionally vapid exercise in mystery and
    ↪ mystification.
```

```
a slightly flashy but narratively opaque and deliberately vapid exercise in style and
    ↪ detail.
```

```
a visually flashy but narratively accessible and emotionally vapid exercise in style and
    ↪ creativity.
```

```
a fairly flashy but narratively vivid and emotionally vapid exercise in style and
    ↪ mystification.
```

```
a somewhat flashy but socially opaque and emotionally vapid exercise in style and
    ↪ technique.
```

```
a fairly flashy but extremely opaque and emotionally vapid exercise in beauty and
    ↪ mystification.
```


```
a little flashy but extremely lively and fairly vapid exercise in relaxation and
    ↪ mystification.
```

```
a slightly flashy but highly opaque and somewhat vapid exercise in movement and
    ↪ breathing.
```

```
a visually flashy but narratively sensitive and emotionally vapid exercise in style and
    ↪ mystification.
```

(continues on next page)

(continued from previous page)

a visually flashy but emotionally opaque and emotionally vapid exercise in beauty and mystery.

Examples where anchor applies and model predicts positive:

8.4 Contrastive Explanation Method

8.4.1 Contrastive Explanations Method (CEM) applied to Iris dataset

The Contrastive Explanation Method (CEM) can generate black box model explanations in terms of pertinent positives (PP) and pertinent negatives (PN). For PP, it finds what should be minimally and sufficiently present (e.g. important pixels in an image) to justify its classification. PN on the other hand identify what should be minimally and necessarily absent from the explained instance in order to maintain the original prediction.

The original paper where the algorithm is based on can be found on [arXiv](#).

This notebook requires the seaborn package for visualization which can be installed via pip:

Note

To enable support for the Contrastive Explanation Method, you may need to run

```
pip install alibi[tensorflow]
```

```
[ ]: !pip install seaborn
```

```
[1]: import tensorflow as tf
tf.get_logger().setLevel(40) # suppress deprecation messages
tf.compat.v1.disable_v2_behavior() # disable TF2 behaviour as alibi code still relies on TF1 constructs
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.utils import to_categorical

import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import seaborn as sns
from sklearn.datasets import load_iris
from alibi.explainers import CEM

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False
```

```
TF version: 2.2.0
Eager execution enabled: False
```

Load and prepare Iris dataset

```
[2]: dataset = load_iris()
feature_names = dataset.feature_names
class_names = list(dataset.target_names)
```

Scale data

```
[3]: dataset.data = (dataset.data - dataset.data.mean(axis=0)) / dataset.data.std(axis=0)
```

Define training and test set

```
[4]: idx = 145
x_train, y_train = dataset.data[:idx, :], dataset.target[:idx]
x_test, y_test = dataset.data[idx+1: :, :], dataset.target[idx+1:]
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Define and train logistic regression model

```
[5]: def lr_model():
    x_in = Input(shape=(4,))
    x_out = Dense(3, activation='softmax')(x_in)
    lr = Model(inputs=x_in, outputs=x_out)
    lr.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
    return lr
```

```
[6]: lr = lr_model()
lr.summary()
lr.fit(x_train, y_train, batch_size=16, epochs=500, verbose=0)
lr.save('iris_lr.h5', save_format='h5')
```

Model: "model"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 4)]	0
dense (Dense)	(None, 3)	15
Total params: 15		
Trainable params: 15		
Non-trainable params: 0		

Generate contrastive explanation with pertinent negative

Explained instance:

```
[7]: idx = 0
X = x_test[idx].reshape((1,) + x_test[idx].shape)
print(f'Prediction on instance to be explained: {class_names[np.argmax(lr.predict(X))]}')
print(f'Prediction probabilities for each class on the instance: {lr.predict(X)}')
```

Prediction on instance to be explained: virginica
 Prediction probabilities for each class on the instance: [[2.2735458e-04 2.4420770e-01 7.
 ↪5556499e-01]]

CEM parameters:

```
[8]: mode = 'PN' # 'PN' (pertinent negative) or 'PP' (pertinent positive)
shape = (1,) + x_train.shape[1:] # instance shape
kappa = .2 # minimum difference needed between the prediction probability for the
↪perturbed instance on the
        # class predicted by the original instance and the max probability on the
↪other classes
        # in order for the first loss term to be minimized
beta = .1 # weight of the L1 loss term
c_init = 10. # initial weight c of the loss term encouraging to predict a different
↪class (PN) or
        # the same class (PP) for the perturbed instance compared to the original
↪instance to be explained
c_steps = 10 # nb of updates for c
max_iterations = 1000 # nb of iterations per value of c
feature_range = (x_train.min(axis=0).reshape(shape)-.1, # feature range for the
↪perturbed instance
                x_train.max(axis=0).reshape(shape)+.1) # can be either a float or
↪array of shape (1xfeatures)
clip = (-1000.,1000.) # gradient clipping
lr_init = 1e-2 # initial learning rate
```

Generate pertinent negative:

```
[9]: # define model
lr = load_model('iris_lr.h5')

# initialize CEM explainer and explain instance
cem = CEM(lr, mode, shape, kappa=kappa, beta=beta, feature_range=feature_range,
          max_iterations=max_iterations, c_init=c_init, c_steps=c_steps,
          learning_rate_init=lr_init, clip=clip)
cem.fit(x_train, no_info_type='median') # we need to define what feature values contain
↪the least
                                     # info wrt predictions
                                     # here we will naively assume that the feature-
↪wise median
                                     # contains no info; domain knowledge helps!
explanation = cem.explain(X, verbose=False)
```



```
[10]: print(f'Original instance: {explanation.X}')
      print(f'Predicted class: {class_names[explanation.X_pred]}')

Original instance: [[ 0.55333328 -1.28296331  0.70592084  0.92230284]]
Predicted class: virginica
```

```
[11]: print(f'Pertinent negative: {explanation.PN}')
      print(f'Predicted class: {class_names[explanation.PN_pred]}')

Pertinent negative: [[ 0.55333333 -1.2829633  -0.5391252  0.92230284]]
Predicted class: versicolor
```

Store explanation to plot later on:

```
[12]: expl = {}
      expl['PN'] = explanation.PN
      expl['PN_pred'] = explanation.PN_pred
```

Generate pertinent positive

```
[13]: mode = 'PP'
```

Generate pertinent positive:

```
[14]: # define model
      lr = load_model('iris_lr.h5')

      # initialize CEM explainer and explain instance
      cem = CEM(lr, mode, shape, kappa=kappa, beta=beta, feature_range=feature_range,
                max_iterations=max_iterations, c_init=c_init, c_steps=c_steps,
                learning_rate_init=lr_init, clip=clip)
      cem.fit(x_train, no_info_type='median')
      explanation = cem.explain(X, verbose=False)
```

```
[15]: print(f'Pertinent positive: {explanation.PP}')
      print(f'Predicted class: {class_names[explanation.PP_pred]}')

Pertinent positive: [[-7.44469730e-09 -3.47054341e-08  2.68840638e-01  9.17062904e-01]]
Predicted class: virginica
```

```
[16]: expl['PP'] = explanation.PP
      expl['PP_pred'] = explanation.PP_pred
```

Visualize PN and PP

Let's visualize the generated explanations to check if the perturbed instances make sense.

Create dataframe from standardized data:

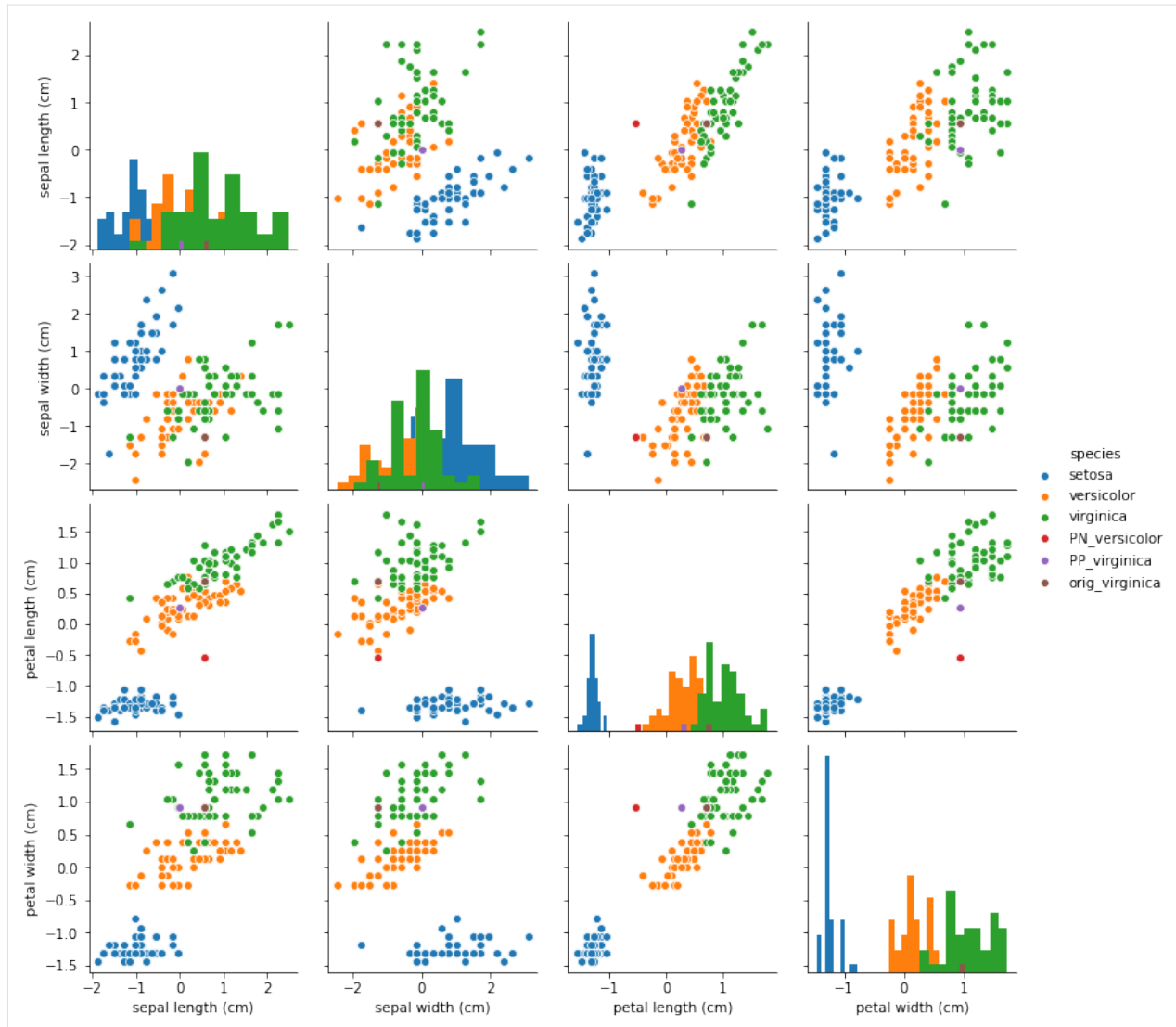
```
[17]: df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
df['species'] = np.array([dataset.target_names[i] for i in dataset.target])
```

Highlight explained instance and add pertinent negative and positive to the dataset:

```
[18]: pn = pd.DataFrame(expl['PN'], columns=dataset.feature_names)
pn['species'] = 'PN_' + class_names[expl['PN_pred']]
pp = pd.DataFrame(expl['PP'], columns=dataset.feature_names)
pp['species'] = 'PP_' + class_names[expl['PP_pred']]
orig_inst = pd.DataFrame(explanation.X, columns=dataset.feature_names)
orig_inst['species'] = 'orig_' + class_names[explanation.X_pred]
df = pd.concat([df, pn, pp, orig_inst], ignore_index=True)
```

Pair plots between the features show that the pertinent negative is pushed from the original instance (versicolor) into the virginica distribution while the pertinent positive moved away from the virginica distribution.

```
[19]: fig = sns.pairplot(df, hue='species', diag_kind='hist');
```



Use numerical gradients in CEM

If we do not have access to the Keras or TensorFlow model weights, we can use numerical gradients for the first term in the loss function that needs to be minimized (eq. 1 and 4 in the [paper](#)).

CEM parameters:

```
[20]: mode = 'PN'
```

If numerical gradients are used to compute:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial p} * \frac{\partial p}{\partial x}$$

with L = loss function; p = predict function and x the parameter to optimize, then the tuple *eps* can be used to define the perturbation used to compute the derivatives. *eps[0]* is used to calculate the first partial derivative term and *eps[1]* is used for the second term. *eps[0]* and *eps[1]* can be a combination of float values or numpy arrays. For *eps[0]*, the array dimension should be $(1 \times nb \text{ of prediction categories})$ and for *eps[1]* it should be $(1 \times nb \text{ of features})$.

```
[21]: eps0 = np.array([[1e-2, 1e-2, 1e-2]]) # 3 prediction categories, equivalent to 1e-2
      eps1 = np.array([[1e-2, 1e-2, 1e-2, 1e-2]]) # 4 features, also equivalent to 1e-2
      eps = (eps0, eps1)
```

For complex models with a high number of parameters and a high dimensional feature space (e.g. Inception on ImageNet), evaluating numerical gradients can be expensive as they involve multiple prediction calls for each perturbed instance. The `update_num_grad` parameter allows you to set a batch size on which to evaluate the numerical gradients, drastically reducing the number of prediction calls required.

```
[22]: update_num_grad = 1
```

Generate pertinent negative:

```
[23]: # define model
      lr = load_model('iris_lr.h5')
      predict_fn = lambda x: lr.predict(x) # only pass the predict fn which takes numpy_
      ↪ arrays to CEM
      # explainer can no longer minimize wrt model_
      ↪ weights

      # initialize CEM explainer and explain instance
      cem = CEM(predict_fn, mode, shape, kappa=kappa, beta=beta,
                 feature_range=feature_range, max_iterations=max_iterations,
                 eps=eps, c_init=c_init, c_steps=c_steps, learning_rate_init=lr_init,
                 clip=clip, update_num_grad=update_num_grad)
      cem.fit(x_train, no_info_type='median')
      explanation = cem.explain(X, verbose=False)
```

```
[24]: print(f'Original instance: {explanation.X}')
      print(f'Predicted class: {class_names[explanation.X_pred]}')

      Original instance: [[ 0.55333328 -1.28296331  0.70592084  0.92230284]]
      Predicted class: virginica
```

```
[25]: print(f'Pertinent negative: {explanation.X}')
      print(f'Predicted class: {class_names[explanation.X_pred]}')

      Pertinent negative: [[ 0.55333328 -1.28296331  0.70592084  0.92230284]]
      Predicted class: virginica
```

Clean up:

```
[26]: os.remove('iris_lr.h5')
```

8.4.2 Contrastive Explanations Method (CEM) applied to MNIST

The Contrastive Explanation Method (CEM) can generate black box model explanations in terms of pertinent positives (PP) and pertinent negatives (PN). For PP, it finds what should be minimally and sufficiently present (e.g. important pixels in an image) to justify its classification. PN on the other hand identify what should be minimally and necessarily absent from the explained instance in order to maintain the original prediction.

The original paper where the algorithm is based on can be found on [arXiv](#).

Note

To enable support for the Contrastive Explanation Method, you may need to run

```
pip install alibi[tensorflow]
```

```
[1]: import tensorflow as tf
tf.get_logger().setLevel(40) # suppress deprecation messages
tf.compat.v1.disable_v2_behavior() # disable TF2 behaviour as alibi code still relies on
↳ TF1 constructs
import tensorflow.keras as keras
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input,
↳ UpSampling2D
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.utils import to_categorical

import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
from alibi.explainers import CEM

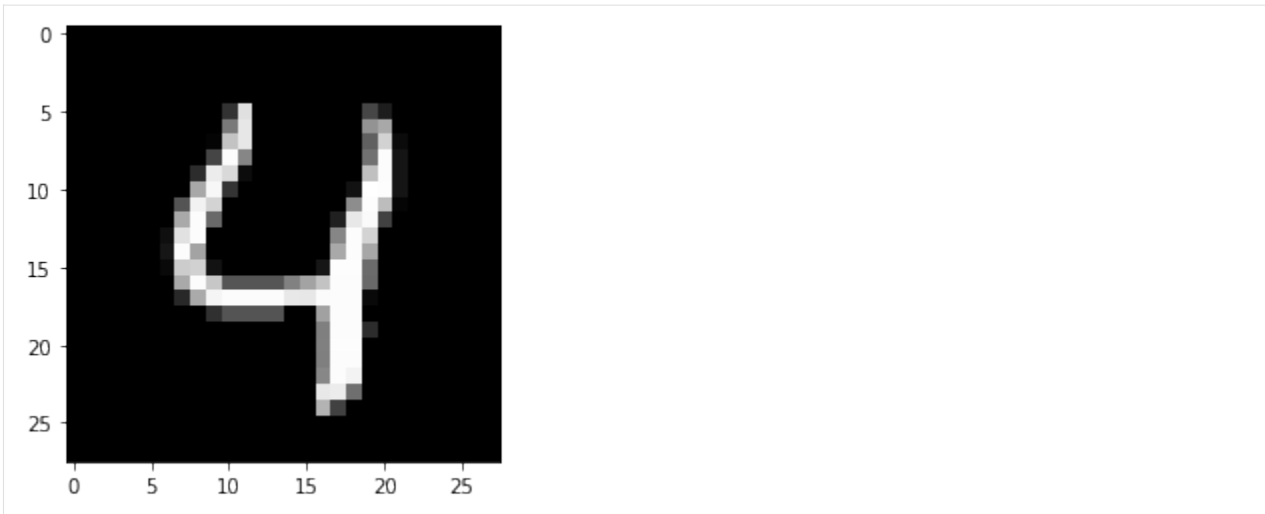
print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False

TF version: 2.2.0
Eager execution enabled: False
```

Load and prepare MNIST data

```
[2]: (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)
plt.gray()
plt.imshow(x_test[4]);

x_train shape: (60000, 28, 28) y_train shape: (60000,)
```



Prepare data: scale, reshape and categorize

```
[3]: x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = np.reshape(x_train, x_train.shape + (1,))
x_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)

x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
y_train shape: (60000, 10) y_test shape: (10000, 10)
```

```
[4]: xmin, xmax = -.5, .5
x_train = ((x_train - x_train.min()) / (x_train.max() - x_train.min())) * (xmax - xmin) + xmin
x_test = ((x_test - x_test.min()) / (x_test.max() - x_test.min())) * (xmax - xmin) + xmin
```

Define and train CNN model

```
[5]: def cnn_model():
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(filters=64, kernel_size=2, padding='same', activation='relu')(x_in)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)

    x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)

    x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)
```

(continues on next page)

(continued from previous page)

```

x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
x_out = Dense(10, activation='softmax')(x)

cnn = Model(inputs=x_in, outputs=x_out)
cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

return cnn

```

```

[6]: cnn = cnn_model()
cnn.summary()
cnn.fit(x_train, y_train, batch_size=64, epochs=5, verbose=1)
cnn.save('mnist_cnn.h5', save_format='h5')

```

Model: "model"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d (Conv2D)	(None, 28, 28, 64)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 14, 14, 32)	8224
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 32)	0
dropout_1 (Dropout)	(None, 7, 7, 32)	0
conv2d_2 (Conv2D)	(None, 7, 7, 32)	4128
max_pooling2d_2 (MaxPooling2D)	(None, 3, 3, 32)	0
dropout_2 (Dropout)	(None, 3, 3, 32)	0
flatten (Flatten)	(None, 288)	0
dense (Dense)	(None, 256)	73984
dropout_3 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2570
Total params: 89,226		
Trainable params: 89,226		
Non-trainable params: 0		

Evaluate the model on test set

```
[7]: cnn = load_model('mnist_cnn.h5')
score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])
```

```
Test accuracy:  0.9871
```

Define and train auto-encoder

```
[8]: def ae_model():
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x_in)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)
    encoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)

    x = Conv2D(16, (3, 3), activation='relu', padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    decoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)

    autoencoder = Model(x_in, decoded)
    autoencoder.compile(optimizer='adam', loss='mse')

    return autoencoder
```

```
[9]: ae = ae_model()
ae.summary()
ae.fit(x_train, x_train, batch_size=128, epochs=4, validation_data=(x_test, x_test),
      verbose=0)
ae.save('mnist_ae.h5', save_format='h5')
```

```
Model: "model_1"
```

Layer (type)	Output Shape	Param #
=====		
input_2 (InputLayer)	[(None, 28, 28, 1)]	0

conv2d_3 (Conv2D)	(None, 28, 28, 16)	160

conv2d_4 (Conv2D)	(None, 28, 28, 16)	2320

max_pooling2d_3 (MaxPooling2D)	(None, 14, 14, 16)	0

conv2d_5 (Conv2D)	(None, 14, 14, 1)	145

conv2d_6 (Conv2D)	(None, 14, 14, 16)	160

up_sampling2d (UpSampling2D)	(None, 28, 28, 16)	0

conv2d_7 (Conv2D)	(None, 28, 28, 16)	2320

conv2d_8 (Conv2D)	(None, 28, 28, 1)	145

(continues on next page)

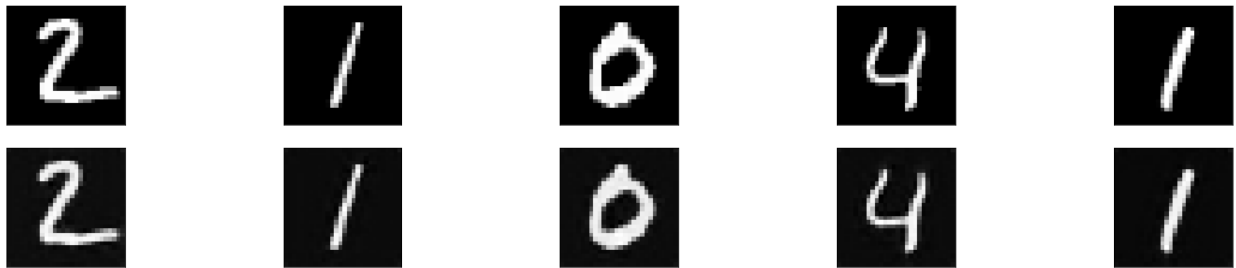
(continued from previous page)

```
=====
Total params: 5,250
Trainable params: 5,250
Non-trainable params: 0
=====
```

Compare original with decoded images

```
[10]: ae = load_model('mnist_ae.h5')

decoded_imgs = ae.predict(x_test)
n = 5
plt.figure(figsize=(20, 4))
for i in range(1, n+1):
    # display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

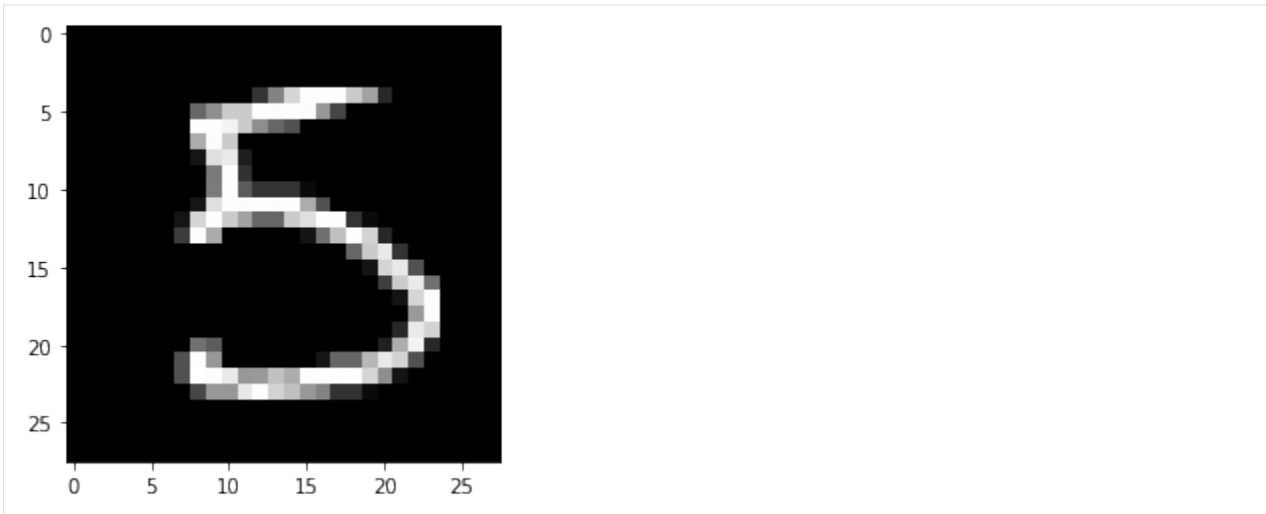


Generate contrastive explanation with pertinent negative

Explained instance:

```
[11]: idx = 15
X = x_test[idx].reshape((1,) + x_test[idx].shape)
```

```
[12]: plt.imshow(X.reshape(28, 28));
```



Model prediction:

```
[13]: cnn.predict(X).argmax(), cnn.predict(X).max()
```

```
[13]: (5, 0.99959975)
```

CEM parameters:

```
[14]: mode = 'PN' # 'PN' (pertinent negative) or 'PP' (pertinent positive)
      shape = (1,) + x_train.shape[1:] # instance shape
      kappa = 0. # minimum difference needed between the prediction probability for the
      ↪ perturbed instance on the
      ↪ class predicted by the original instance and the max probability on the
      ↪ other classes
      # in order for the first loss term to be minimized
      beta = .1 # weight of the L1 loss term
      gamma = 100 # weight of the optional auto-encoder loss term
      c_init = 1. # initial weight c of the loss term encouraging to predict a different
      ↪ class (PN) or
      ↪ the same class (PP) for the perturbed instance compared to the original
      ↪ instance to be explained
      c_steps = 10 # nb of updates for c
      max_iterations = 1000 # nb of iterations per value of c
      feature_range = (x_train.min(), x_train.max()) # feature range for the perturbed instance
      clip = (-1000., 1000.) # gradient clipping
      lr = 1e-2 # initial learning rate
      no_info_val = -1. # a value, float or feature-wise, which can be seen as containing no
      ↪ info to make a prediction
      ↪ # perturbations towards this value means removing features, and away
      ↪ means adding features
      # for our MNIST images, the background (-0.5) is the least informative,
      # so positive/negative perturbations imply adding/removing features
```

Generate pertinent negative:

```
[15]: # initialize CEM explainer and explain instance
      cem = CEM(cnn, mode, shape, kappa=kappa, beta=beta, feature_range=feature_range,
```

(continues on next page)

(continued from previous page)

```

        gamma=gamma, ae_model=ae, max_iterations=max_iterations,
        c_init=c_init, c_steps=c_steps, learning_rate_init=lr, clip=clip, no_info_
        ↪ val=no_info_val)

explanation = cem.explain(X)

```

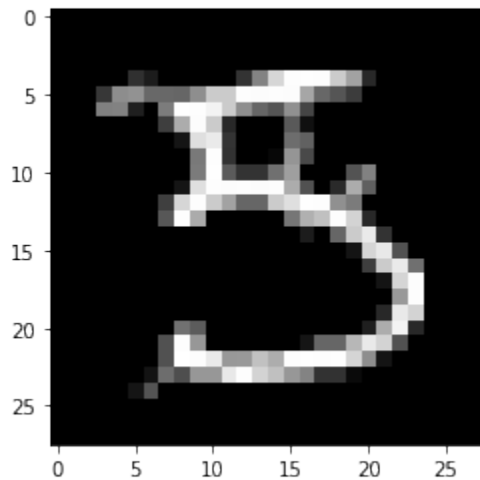
Pertinent negative:

```

[16]: print(f'Pertinent negative prediction: {explanation.PN_pred}')
      plt.imshow(explanation.PN.reshape(28, 28));

```

Pertinent negative prediction: 3



Generate pertinent positive

```

[17]: mode = 'PP'

```

```

[18]: # initialize CEM explainer and explain instance
cem = CEM(cnn, mode, shape, kappa=kappa, beta=beta, feature_range=feature_range,
        gamma=gamma, ae_model=ae, max_iterations=max_iterations,
        c_init=c_init, c_steps=c_steps, learning_rate_init=lr, clip=clip, no_info_
        ↪ val=no_info_val)

explanation = cem.explain(X)

```

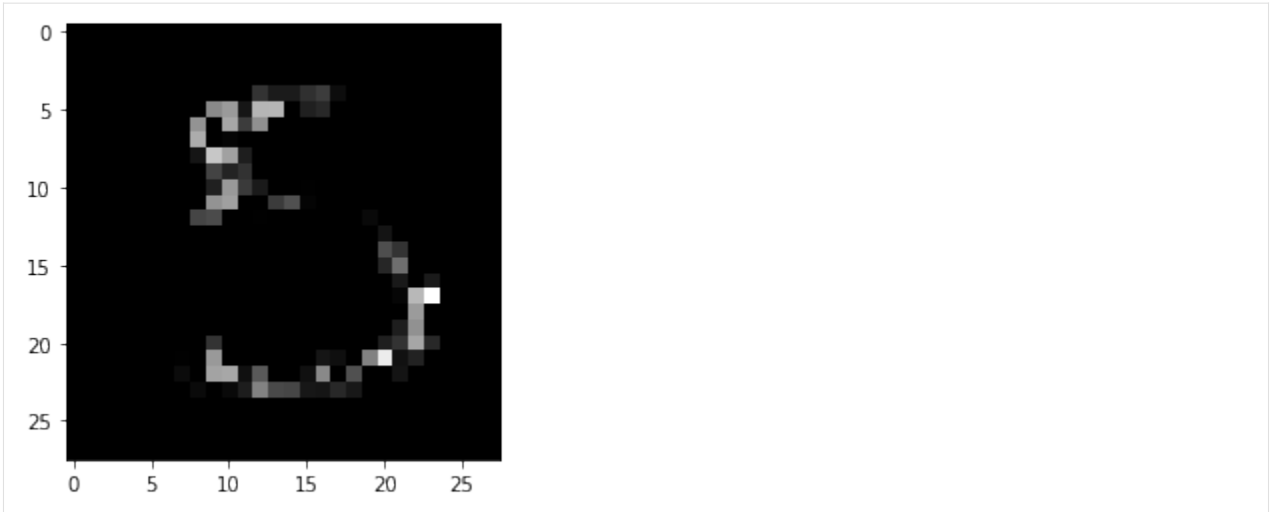
Pertinent positive:

```

[19]: print(f'Pertinent positive prediction: {explanation.PP_pred}')
      plt.imshow(explanation.PP.reshape(28, 28));

```

Pertinent positive prediction: 5



Clean up:

```
[ ]: os.remove('mnist_cnn.h5')
os.remove('mnist_ae.h5')
```

8.5 Counterfactual Instances

8.5.1 Counterfactual instances on MNIST

Given a test instance X , this method can generate counterfactual instances X' given a desired counterfactual class t which can either be a class specified upfront or any other class that is different from the predicted class of X .

The loss function for finding counterfactuals is the following:

$$L(X'|X) = (f_t(X') - p_t)^2 + \lambda L_1(X', X).$$

The first loss term, guides the search towards instances X' for which the predicted class probability $f_t(X')$ is close to a pre-specified target class probability p_t (typically $p_t = 1$). The second loss term ensures that the counterfactuals are close in the feature space to the original test instance.

In this notebook we illustrate the usage of the basic counterfactual algorithm on the MNIST dataset.

Note

To enable support for Counterfactual, you may need to run

```
pip install alibi[tensorflow]
```

```
[1]: import tensorflow as tf
tf.get_logger().setLevel(40) # suppress deprecation messages
tf.compat.v1.disable_v2_behavior() # disable TF2 behaviour as alibi code still relies on
↳ TF1 constructs
from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input
from tensorflow.keras.models import Model, load_model
```

(continues on next page)

(continued from previous page)

```

from tensorflow.keras.utils import to_categorical
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
from time import time
from alibi.explainers import Counterfactual
print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False

```

```

TF version: 2.2.0
Eager execution enabled: False

```

Load and prepare MNIST data

```

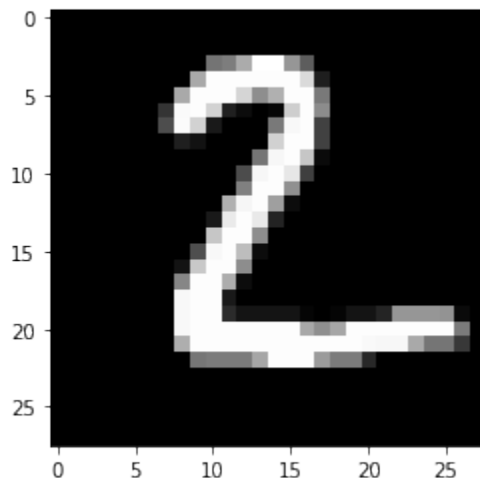
[2]: (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)
plt.gray()
plt.imshow(x_test[1]);

```

```

x_train shape: (60000, 28, 28) y_train shape: (60000,)

```



Prepare data: scale, reshape and categorize

```

[3]: x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = np.reshape(x_train, x_train.shape + (1,))
x_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)

```

```

x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
y_train shape: (60000, 10) y_test shape: (10000, 10)

```

```
[4]: xmin, xmax = -.5, .5
x_train = ((x_train - x_train.min()) / (x_train.max() - x_train.min())) * (xmax - xmin) + xmin
x_test = ((x_test - x_test.min()) / (x_test.max() - x_test.min())) * (xmax - xmin) + xmin
```

Define and train CNN model

```
[5]: def cnn_model():
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(filters=64, kernel_size=2, padding='same', activation='relu')(x_in)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)

    x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)

    x = Flatten()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.5)(x)
    x_out = Dense(10, activation='softmax')(x)

    cnn = Model(inputs=x_in, outputs=x_out)
    cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    return cnn
```

```
[6]: cnn = cnn_model()
cnn.summary()
cnn.fit(x_train, y_train, batch_size=64, epochs=3, verbose=0)
cnn.save('mnist_cnn.h5')
```

Model: "model"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d (Conv2D)	(None, 28, 28, 64)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 14, 14, 32)	8224
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 32)	0
dropout_1 (Dropout)	(None, 7, 7, 32)	0
flatten (Flatten)	(None, 1568)	0

(continues on next page)

(continued from previous page)

dense (Dense)	(None, 256)	401664
dropout_2 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2570

```

=====
Total params: 412,778
Trainable params: 412,778
Non-trainable params: 0
=====

```

Evaluate the model on test set

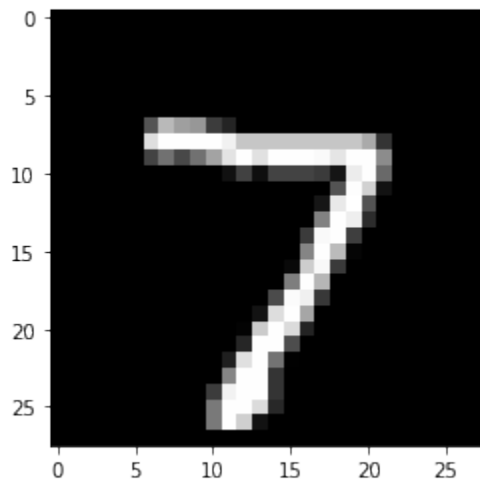
```
[7]: cnn = load_model('mnist_cnn.h5')
score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])
```

Test accuracy: 0.9871

Generate counterfactuals

Original instance:

```
[8]: X = x_test[0].reshape((1,) + x_test[0].shape)
plt.imshow(X.reshape(28, 28));
```



Counterfactual parameters:

```
[9]: shape = (1,) + x_train.shape[1:]
target_proba = 1.0
tol = 0.01 # want counterfactuals with p(class)>0.99
target_class = 'other' # any class other than 7 will do
max_iter = 1000
lam_init = 1e-1
max_lam_steps = 10
```

(continues on next page)

(continued from previous page)

```
learning_rate_init = 0.1
feature_range = (x_train.min(), x_train.max())
```

Run counterfactual:

```
[10]: # initialize explainer
cf = Counterfactual(cnn, shape=shape, target_proba=target_proba, tol=tol,
                    target_class=target_class, max_iter=max_iter, lam_init=lam_init,
                    max_lam_steps=max_lam_steps, learning_rate_init=learning_rate_init,
                    feature_range=feature_range)

start_time = time()
explanation = cf.explain(X)
print('Explanation took {:.3f} sec'.format(time() - start_time))

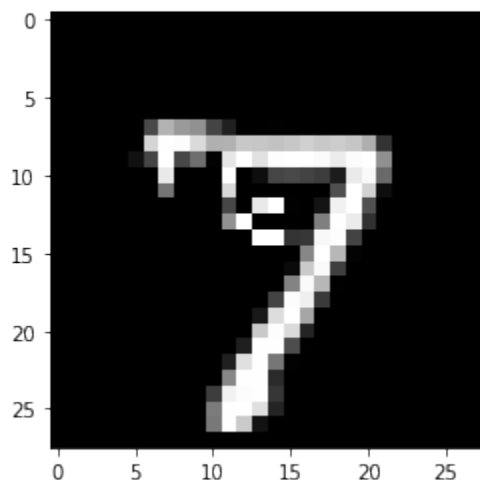
Explanation took 8.407 sec
```

Results:

```
[11]: pred_class = explanation.cf['class']
proba = explanation.cf['proba'][0][pred_class]

print(f'Counterfactual prediction: {pred_class} with probability {proba}')
plt.imshow(explanation.cf['X'].reshape(28, 28));
```

Counterfactual prediction: 9 with probability 0.9924006462097168



The counterfactual starting from a 7 moves towards the closest class as determined by the model and the data - in this case a 9. The evolution of the counterfactual during the iterations over λ can be seen below (note that all of the following examples satisfy the counterfactual condition):

```
[12]: n_cfs = np.array([len(explanation.all[iter_cf]) for iter_cf in range(max_lam_steps)])
examples = {}
for ix, n in enumerate(n_cfs):
    if n>0:
        examples[ix] = {'ix': ix, 'lambda': explanation.all[ix][0]['lambda'],
                        'X': explanation.all[ix][0]['X']}
```

(continues on next page)

(continued from previous page)

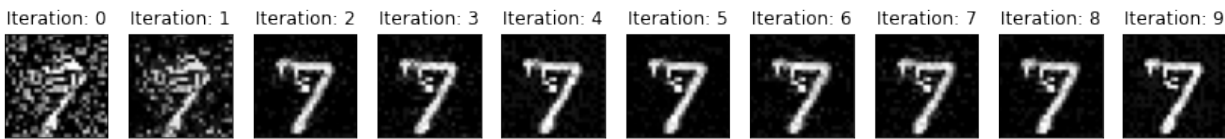
```

columns = len(examples) + 1
rows = 1

fig = plt.figure(figsize=(16,6))

for i, key in enumerate(examples.keys()):
    ax = plt.subplot(rows, columns, i+1)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.imshow(examples[key]['X'].reshape(28,28))
    plt.title(f'Iteration: {key}')

```



Typically, the first few iterations find counterfactuals that are out of distribution, while the later iterations make the counterfactual more sparse and interpretable.

Let's now try to steer the counterfactual to a specific class:

```

[13]: target_class = 1

cf = Counterfactual(cnn, shape=shape, target_proba=target_proba, tol=tol,
                    target_class=target_class, max_iter=max_iter, lam_init=lam_init,
                    max_lam_steps=max_lam_steps, learning_rate_init=learning_rate_init,
                    feature_range=feature_range)

explanation = start_time = time()
explanation = cf.explain(X)
print('Explanation took {:.3f} sec'.format(time() - start_time))

```

Explanation took 6.249 sec

Results:

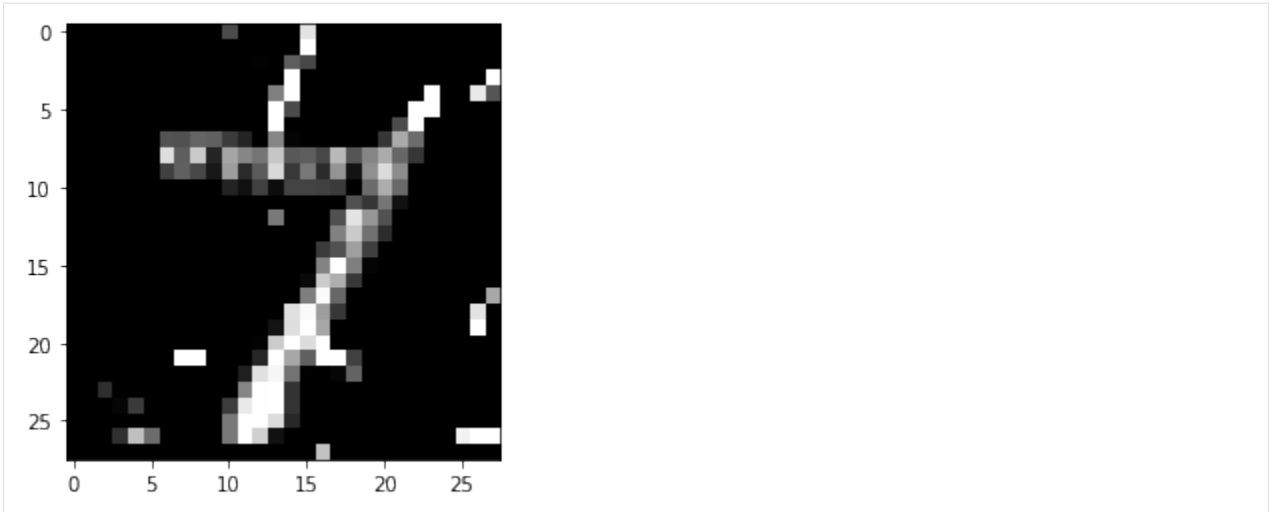
```

[14]: pred_class = explanation.cf['class']
      proba = explanation.cf['proba'][0][pred_class]

      print(f'Counterfactual prediction: {pred_class} with probability {proba}')
      plt.imshow(explanation.cf['X'].reshape(28, 28));

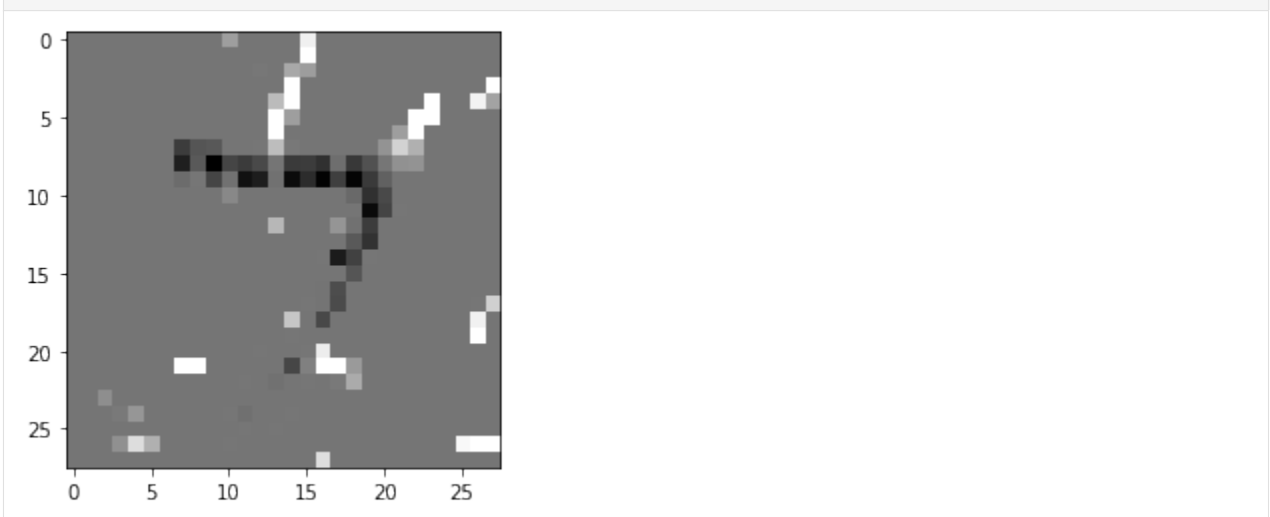
```

Counterfactual prediction: 1 with probability 0.9999160766601562



As you can see, by specifying a class, the search process can't go towards the closest class to the test instance (in this case a 9 as we saw previously), so the resulting counterfactual might be less interpretable. We can gain more insight by looking at the difference between the counterfactual and the original instance:

```
[15]: plt.imshow((explanation.cf['X'] - X).reshape(28, 28));
```



This shows that the counterfactual is stripping out the top part of the 7 to make to result in a prediction of 1 - not very surprising as the dataset has a lot of examples of diagonally slanted 1's.

Clean up:

```
[16]: os.remove('mnist_cnn.h5')
```

8.6 Counterfactuals Guided by Prototypes

8.6.1 Counterfactual explanations with one-hot encoded categorical variables

Real world machine learning applications often handle data with categorical variables. Explanation methods which rely on perturbations of the input features need to make sure those perturbations are meaningful and capture the underlying structure of the data. This becomes tricky for categorical features. For instance random perturbations across possible categories or enforcing a ranking between categories based on frequency of occurrence in the training data do not capture this structure. Our method captures the relation between categories of a variable numerically through the context given by the other features in the data and/or the predictions made by the model. First it captures the pairwise distances between categories and then applies multi-dimensional scaling. More details about the method can be found in the [documentation](#). The example notebook illustrates this approach on the *adult* dataset, which contains a mixture of categorical and numerical features used to predict whether a person's income is above or below \$50k.

Note

To enable support for CounterfactualProto, you may need to run

```
pip install alibi[tensorflow]
```

```
[1]: import tensorflow as tf
tf.get_logger().setLevel(40) # suppress deprecation messages
tf.compat.v1.disable_v2_behavior() # disable TF2 behaviour as alibi code still relies on
↳ TF1 constructs
from tensorflow.keras.layers import Dense, Dropout, Input
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical

%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import os
from sklearn.preprocessing import OneHotEncoder
from time import time
from alibi.datasets import fetch_adult
from alibi.explainers import CounterfactualProto
from alibi.utils import ohe_to_ord, ord_to_ohe

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False

TF version: 2.8.0
Eager execution enabled: False
```

Load adult dataset

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of the categories in each categorical variable.

```
[2]: adult = fetch_adult()
data = adult.data
target = adult.target
feature_names = adult.feature_names
category_map_tmp = adult.category_map
target_names = adult.target_names
```

Define shuffled training and test set:

```
[3]: def set_seed(s=0):
    np.random.seed(s)
    tf.random.set_seed(s)
```

```
[4]: set_seed()
data_perm = np.random.permutation(np.c_[data, target])
X = data_perm[:, :-1]
y = data_perm[:, -1]
```

```
[5]: idx = 30000
y_train, y_test = y[:idx], y[idx+1:]
```

Reorganize data so categorical features come first:

```
[6]: X = np.c_[X[:, 1:8], X[:, 11], X[:, 0], X[:, 8:11]]
```

Adjust `feature_names` and `category_map` as well:

```
[7]: feature_names = feature_names[1:8] + feature_names[11:12] + feature_names[0:1] + feature_
    ↪ names[8:11]
print(feature_names)

['Workclass', 'Education', 'Marital Status', 'Occupation', 'Relationship', 'Race', 'Sex',
 ↪ 'Country', 'Age', 'Capital Gain', 'Capital Loss', 'Hours per week']
```

```
[8]: category_map = {}
for i, (_, v) in enumerate(category_map_tmp.items()):
    category_map[i] = v
```

Create a dictionary with as keys the categorical columns and values the number of categories for each variable in the dataset:

```
[9]: cat_vars_ord = {}
n_categories = len(list(category_map.keys()))
for i in range(n_categories):
    cat_vars_ord[i] = len(np.unique(X[:, i]))
print(cat_vars_ord)

{0: 9, 1: 7, 2: 4, 3: 9, 4: 6, 5: 5, 6: 2, 7: 11}
```

Since we will apply one-hot encoding (OHE) on the categorical variables, we convert `cat_vars_ord` from the ordinal to OHE format. `alibi.utils.mapping` contains utility functions to do this. The keys in `cat_vars_ohe` now represent the first column index for each one-hot encoded categorical variable. This dictionary will later be used in the counterfactual explanation.

```
[10]: cat_vars_ohe = ord_to_ohe(X, cat_vars_ord)[1]
      print(cat_vars_ohe)

{0: 9, 9: 7, 16: 4, 20: 9, 29: 6, 35: 5, 40: 2, 42: 11}
```

Preprocess data

Scale numerical features between -1 and 1:

```
[11]: X_num = X[:, -4:].astype(np.float32, copy=False)
      xmin, xmax = X_num.min(axis=0), X_num.max(axis=0)
      rng = (-1., 1.)
      X_num_scaled = (X_num - xmin) / (xmax - xmin) * (rng[1] - rng[0]) + rng[0]
```

Apply OHE to categorical variables:

```
[12]: X_cat = X[:, :-4].copy()
      ohe = OneHotEncoder(categories='auto', sparse_output=False).fit(X_cat)
      X_cat_ohe = ohe.transform(X_cat)
```

Combine numerical and categorical data:

```
[13]: X = np.c_[X_cat_ohe, X_num_scaled].astype(np.float32, copy=False)
      X_train, X_test = X[:idx, :], X[idx+1:, :]
      print(X_train.shape, X_test.shape)

(30000, 57) (2560, 57)
```

Train neural net

```
[14]: def nn_ohe():

      x_in = Input(shape=(57,))
      x = Dense(60, activation='relu')(x_in)
      x = Dropout(.2)(x)
      x = Dense(60, activation='relu')(x)
      x = Dropout(.2)(x)
      x = Dense(60, activation='relu')(x)
      x = Dropout(.2)(x)
      x_out = Dense(2, activation='softmax')(x)

      nn = Model(inputs=x_in, outputs=x_out)
      nn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

      return nn
```

```
[15]: set_seed()
nn = nn_ohe()
nn.summary()
nn.fit(X_train, to_categorical(y_train), batch_size=256, epochs=30, verbose=0)
```

Model: "model"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 57)]	0
dense (Dense)	(None, 60)	3480
dropout (Dropout)	(None, 60)	0
dense_1 (Dense)	(None, 60)	3660
dropout_1 (Dropout)	(None, 60)	0
dense_2 (Dense)	(None, 60)	3660
dropout_2 (Dropout)	(None, 60)	0
dense_3 (Dense)	(None, 2)	122
Total params: 10,922		
Trainable params: 10,922		
Non-trainable params: 0		

```
[15]: <tensorflow.python.keras.callbacks.History at 0x7f2d7cc5e410>
```

Generate counterfactual

Original instance:

```
[16]: X = X_test[0].reshape((1,) + X_test[0].shape)
```

Initialize counterfactual parameters. The feature perturbations are applied in the numerical feature space, after transforming the categorical variables to numerical features. As a result, the dimensionality and values of `feature_range` are defined in the numerical space.

```
[17]: shape = X.shape
beta = .01
c_init = 1.
c_steps = 5
max_iterations = 500
rng = (-1., 1.) # scale features between -1 and 1
rng_shape = (1,) + data.shape[1:]
feature_range = ((np.ones(rng_shape) * rng[0]).astype(np.float32),
                 (np.ones(rng_shape) * rng[1]).astype(np.float32))
```

Initialize explainer:

```
[18]: def set_seed(s=0):
      np.random.seed(s)
      tf.random.set_seed(s)
```

```
[19]: set_seed()
      cf = CounterfactualProto(nm,
                              shape,
                              beta=beta,
                              cat_vars=cat_vars_ohe,
                              ohe=True, # OHE flag
                              max_iterations=max_iterations,
                              feature_range=feature_range,
                              c_init=c_init,
                              c_steps=c_steps
                              )
```

Fit explainer. `d_type` refers to the distance metric used to convert the categorical to numerical values. Valid options are `abdm`, `mvdn` and `abdm-mvdn`. `abdm` infers the distance between categories of the same variable from the context provided by the other variables. This requires binning of the numerical features as well. `mvdn` computes the distance using the model predictions, and `abdm-mvdn` combines both methods. More info on both distance measures can be found in the [documentation](#).

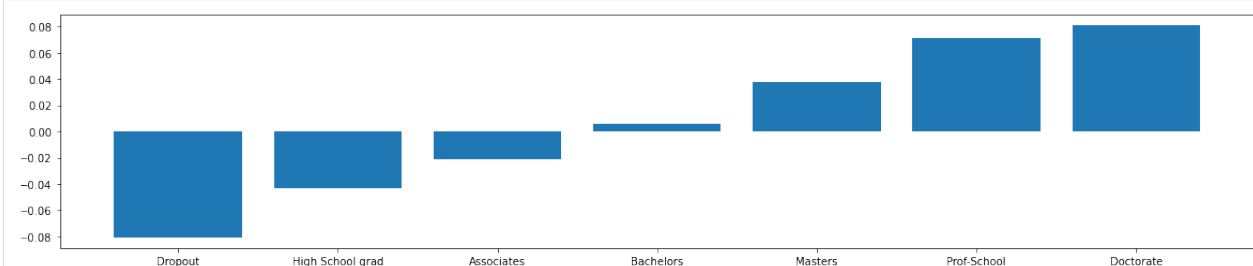
```
[20]: cf.fit(X_train, d_type='abdm', disc_perc=[25, 50, 75]);
```

We can now visualize the transformation from the categorical to numerical values for each category. The example below shows that the **Education** feature is ordered from *High School Dropout* to having obtained a *Doctorate* degree. As a result, if we perturb an instance representing a person that has obtained a *Bachelors* degree, the nearest perturbations will result in a counterfactual instance with either a *Masters* or an *Associates* degree.

```
[21]: def plot_bar(dist, cols, figsize=(10,4)):
      dist = dist.reshape(dist.shape[0])
      idx = np.argsort(dist)
      fig, ax = plt.subplots(figsize=figsize)
      plt.bar(cols[idx], dist[idx])
      print(cols[idx])
```

```
[22]: cat = 'Education'
      idx = feature_names.index(cat)
      plot_bar(cf.d_abs[idx], np.array(category_map[idx]), figsize=(20,4))
```

```
['Dropout' 'High School grad' 'Associates' 'Bachelors' 'Masters'
 'Prof-School' 'Doctorate']
```



Explain instance:

```
[23]: explanation = cf.explain(X)
```

Helper function to more clearly describe explanations:

```
[24]: def describe_instance(X, explanation, eps=1e-2):
    print('Original instance: {} -- proba: {}'.format(target_names[explanation.orig_
    ↪class],
                                                    explanation.orig_proba[0]))
    print('Counterfactual instance: {} -- proba: {}'.format(target_names[explanation.cf[
    ↪'class']],
                                                    explanation.cf['proba'][0]))

    print('\nCounterfactual perturbations...')
    print('\nCategorical:')
    X_orig_ord = ohe_to_ord(X, cat_vars_ohe)[0]
    X_cf_ord = ohe_to_ord(explanation.cf['X'], cat_vars_ohe)[0]
    delta_cat = {}
    for i, (_, v) in enumerate(category_map.items()):
        cat_orig = v[int(X_orig_ord[0, i])]
        cat_cf = v[int(X_cf_ord[0, i])]
        if cat_orig != cat_cf:
            delta_cat[feature_names[i]] = [cat_orig, cat_cf]
    if delta_cat:
        for k, v in delta_cat.items():
            print('{}: {} --> {}'.format(k, v[0], v[1]))
    print('\nNumerical:')
    delta_num = X_cf_ord[0, -4:] - X_orig_ord[0, -4:]
    n_keys = len(list(cat_vars_ord.keys()))
    for i in range(delta_num.shape[0]):
        if np.abs(delta_num[i]) > eps:
            print('{}: {:.2f} --> {:.2f}'.format(feature_names[i+n_keys],
                                                    X_orig_ord[0,i+n_keys],
                                                    X_cf_ord[0,i+n_keys]))
```

```
[25]: describe_instance(X, explanation)
```

```
Original instance: <=50K -- proba: [0.70744723 0.29255277]
Counterfactual instance: >50K -- proba: [0.37736374 0.62263626]

Counterfactual perturbations...

Categorical:
Education: Associates --> Bachelors

Numerical:
```

By obtaining a higher level of education the income is predicted to be above \$50k.

Change the categorical distance metric

Instead of `abdm`, we now use `mvdm` as our distance metric.

```
[26]: set_seed()
      cf.fit(X_train, d_type='mvdm')
      explanation = cf.explain(X)
      describe_instance(X, explanation)

Original instance: <=50K -- proba: [0.70744723 0.29255277]
Counterfactual instance: >50K -- proba: [0.38161737 0.61838263]

Counterfactual perturbations...

Categorical:
Education: Associates --> Bachelors

Numerical:
```

The same conclusion hold using a different distance metric.

Use k-d trees to build prototypes

We can also use *k-d trees* to build class prototypes to guide the counterfactual to nearby instances in the counterfactual class as described in [Interpretable Counterfactual Explanations Guided by Prototypes](#).

```
[27]: use_kdtree = True
      theta = 10. # weight of prototype loss term
```

Initialize, fit and explain instance:

```
[28]: set_seed()
      X = X_test[7].reshape((1,) + X_test[0].shape)
      cf = CounterfactualProto(nn,
                              shape,
                              beta=beta,
                              theta=theta,
                              cat_vars=cat_vars_ohe,
                              ohe=True,
                              use_kdtree=use_kdtree,
                              max_iterations=max_iterations,
                              feature_range=feature_range,
                              c_init=c_init,
                              c_steps=c_steps
                              )
      cf.fit(X_train, d_type='abdm')
      explanation = cf.explain(X)
      describe_instance(X, explanation)

Original instance: <=50K -- proba: [0.5211548 0.47884512]
Counterfactual instance: >50K -- proba: [0.49958408 0.500416 ]

Counterfactual perturbations...
```

(continues on next page)

(continued from previous page)

Categorical:

Numerical:

Age: -0.53 --> -0.51

By slightly increasing the age of the person the income would be predicted to be above \$50k.

Use an autoencoder to build prototypes

Another option is to use an autoencoder to guide the perturbed instance to the counterfactual class. We define and train the autoencoder:

```
[29]: def ae_model():
    # encoder
    x_in = Input(shape=(57,))
    x = Dense(60, activation='relu')(x_in)
    x = Dense(30, activation='relu')(x)
    x = Dense(15, activation='relu')(x)
    encoded = Dense(10, activation=None)(x)
    encoder = Model(x_in, encoded)

    # decoder
    dec_in = Input(shape=(10,))
    x = Dense(15, activation='relu')(dec_in)
    x = Dense(30, activation='relu')(x)
    x = Dense(60, activation='relu')(x)
    decoded = Dense(57, activation=None)(x)
    decoder = Model(dec_in, decoded)

    # autoencoder = encoder + decoder
    x_out = decoder(encoder(x_in))
    autoencoder = Model(x_in, x_out)
    autoencoder.compile(optimizer='adam', loss='mse')

    return autoencoder, encoder, decoder
```

```
[30]: set_seed()
ae, enc, dec = ae_model()
ae.summary()
ae.fit(X_train, X_train, batch_size=128, epochs=100, validation_data=(X_test, X_test),
      verbose=0)
```

Model: "model_3"

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 57)]	0
model_1 (Model)	(None, 10)	5935
model_2 (Model)	(None, 57)	5982

(continues on next page)

(continued from previous page)

```
Total params: 11,917
Trainable params: 11,917
Non-trainable params: 0
```

```
[30]: <tensorflow.python.keras.callbacks.History at 0x7f2d783aff90>
```

Weights for the autoencoder and prototype loss terms:

```
[31]: beta = .1 # L1
      gamma = 10. # autoencoder
      theta = .1 # prototype
```

Initialize, fit and explain instance:

```
[32]: set_seed()
      X = X_test[19].reshape((1,) + X_test[0].shape)
      cf = CounterfactualProto(nm,
                              shape,
                              beta=beta,
                              enc_model=enc,
                              ae_model=ae,
                              gamma=gamma,
                              theta=theta,
                              cat_vars=cat_vars_ohe,
                              ohe=True,
                              max_iterations=max_iterations,
                              feature_range=feature_range,
                              c_init=c_init,
                              c_steps=c_steps
                              )
      cf.fit(X_train, d_type='abdm')
      explanation = cf.explain(X)
      describe_instance(X, explanation)
```

```
Original instance: >50K -- proba: [0.48656026 0.5134398 ]
Counterfactual instance: <=50K -- proba: [0.71456206 0.28543794]
```

Counterfactual perturbations...

Categorical:

Education: High School grad --> Dropout

Numerical:

Black box model with k-d trees

Now we assume that we only have access to the model's prediction function and treat it as a black box. The k-d trees are again used to define the prototypes.

```
[33]: use_kdtree = True
      theta = 10. # weight of prototype loss term
```

Initialize, fit and explain instance:

```
[34]: set_seed()

X = X_test[24].reshape((1,) + X_test[0].shape)

# define predict function
predict_fn = lambda x: nn.predict(x)

cf = CounterfactualProto(predict_fn,
                        shape,
                        beta=beta,
                        theta=theta,
                        cat_vars=cat_vars_ohe,
                        ohe=True,
                        use_kdtree=use_kdtree,
                        max_iterations=max_iterations,
                        feature_range=feature_range,
                        c_init=c_init,
                        c_steps=c_steps
                        )
cf.fit(X_train, d_type='abdm')
explanation = cf.explain(X)
describe_instance(X, explanation)

Original instance: >50K -- proba: [0.20676644 0.7932335 ]
Counterfactual instance: <=50K -- proba: [0.5048416 0.49515834]

Counterfactual perturbations...

Categorical:

Numerical:
Age: -0.15 --> -0.19
Hours per week: -0.20 --> -0.51
```

If the person was younger and worked less, he or she would have a predicted income below \$50k.

8.6.2 Counterfactual explanations with ordinally encoded categorical variables

This example notebook illustrates how to obtain [counterfactual explanations](#) for instances with a mixture of ordinally encoded categorical and numerical variables. A more elaborate notebook highlighting additional functionality can be found [here](#). We generate counterfactuals for instances in the *adult* dataset where we predict whether a person's income is above or below \$50k.

Note

To enable support for CounterfactualProto, you may need to run

```
pip install alibi[tensorflow]
```

```
[1]: import tensorflow as tf
tf.get_logger().setLevel(40) # suppress deprecation messages
tf.compat.v1.disable_v2_behavior() # disable TF2 behaviour as alibi code still relies on
↳TF1 constructs
from tensorflow.keras.layers import Dense, Input, Embedding, Concatenate, Reshape,
↳Dropout, Lambda
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical

%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import os
from sklearn.preprocessing import OneHotEncoder
from time import time
from alibi.datasets import fetch_adult
from alibi.explainers import CounterfactualProto

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False

TF version:  2.2.0
Eager execution enabled:  False
```

Load adult dataset

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of the categories in each categorical variable.

```
[2]: adult = fetch_adult()
data = adult.data
target = adult.target
feature_names = adult.feature_names
category_map_tmp = adult.category_map
target_names = adult.target_names
```

Define shuffled training and test set:

```
[3]: def set_seed(s=0):  
      np.random.seed(s)  
      tf.random.set_seed(s)
```

```
[4]: set_seed()  
data_perm = np.random.permutation(np.c_[data, target])  
X = data_perm[:, :-1]  
y = data_perm[:, -1]
```

```
[5]: idx = 30000  
y_train, y_test = y[idx], y[idx+1:]
```

Reorganize data so categorical features come first:

```
[6]: X = np.c_[X[:, 1:8], X[:, 11], X[:, 0], X[:, 8:11]]
```

Adjust feature_names and category_map as well:

```
[7]: feature_names = feature_names[1:8] + feature_names[11:12] + feature_names[0:1] + feature_  
      ↪ names[8:11]  
print(feature_names)  
  
['Workclass', 'Education', 'Marital Status', 'Occupation', 'Relationship', 'Race', 'Sex',  
 ↪ 'Country', 'Age', 'Capital Gain', 'Capital Loss', 'Hours per week']
```

```
[8]: category_map = {}  
for i, (_, v) in enumerate(category_map_tmp.items()):  
    category_map[i] = v
```

Create a dictionary with as keys the categorical columns and values the number of categories for each variable in the dataset. This dictionary will later be used in the counterfactual explanation.

```
[9]: cat_vars_ord = {}  
n_categories = len(list(category_map.keys()))  
for i in range(n_categories):  
    cat_vars_ord[i] = len(np.unique(X[:, i]))  
print(cat_vars_ord)  
  
{0: 9, 1: 7, 2: 4, 3: 9, 4: 6, 5: 5, 6: 2, 7: 11}
```

Preprocess data

Scale numerical features between -1 and 1:

```
[10]: X_num = X[:, -4:].astype(np.float32, copy=False)  
xmin, xmax = X_num.min(axis=0), X_num.max(axis=0)  
rng = (-1., 1.)  
X_num_scaled = (X_num - xmin) / (xmax - xmin) * (rng[1] - rng[0]) + rng[0]  
X_num_scaled_train = X_num_scaled[idx, :]  
X_num_scaled_test = X_num_scaled[idx+1:, :]
```

Combine numerical and categorical data:

```
[11]: X = np.c_[X[:, :-4], X_num_scaled].astype(np.float32, copy=False)
      X_train, X_test = X[idx, :], X[idx+1:, :]
      print(X_train.shape, X_test.shape)

(30000, 12) (2560, 12)
```

Train a neural net

The neural net will use entity embeddings for the categorical variables.

```
[12]: def nn_ord():

      x_in = Input(shape=(12,))
      layers_in = []

      # embedding layers
      for i, (_, v) in enumerate(cat_vars_ord.items()):
          emb_in = Lambda(lambda x: x[:, i:i+1])(x_in)
          emb_dim = int(max(min(np.ceil(.5 * v), 50), 2))
          emb_layer = Embedding(input_dim=v+1, output_dim=emb_dim, input_length=1)(emb_in)
          emb_layer = Reshape(target_shape=(emb_dim,))(emb_layer)
          layers_in.append(emb_layer)

      # numerical layers
      num_in = Lambda(lambda x: x[:, -4:])(x_in)
      num_layer = Dense(16)(num_in)
      layers_in.append(num_layer)

      # combine
      x = Concatenate()(layers_in)
      x = Dense(60, activation='relu')(x)
      x = Dropout(.2)(x)
      x = Dense(60, activation='relu')(x)
      x = Dropout(.2)(x)
      x = Dense(60, activation='relu')(x)
      x = Dropout(.2)(x)
      x_out = Dense(2, activation='softmax')(x)

      nn = Model(inputs=x_in, outputs=x_out)
      nn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

      return nn
```

```
[13]: set_seed()
      nn = nn_ord()
      nn.summary()
      nn.fit(X_train, to_categorical(y_train), batch_size=128, epochs=30, verbose=0)
```

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
--------------	--------------	---------	--------------

(continues on next page)

(continued from previous page)

input_1 (InputLayer)	[(None, 12)]	0	
↳ ----- lambda (Lambda)	(None, 1)	0	input_1[0][0]
↳ ----- lambda_1 (Lambda)	(None, 1)	0	input_1[0][0]
↳ ----- lambda_2 (Lambda)	(None, 1)	0	input_1[0][0]
↳ ----- lambda_3 (Lambda)	(None, 1)	0	input_1[0][0]
↳ ----- lambda_4 (Lambda)	(None, 1)	0	input_1[0][0]
↳ ----- lambda_5 (Lambda)	(None, 1)	0	input_1[0][0]
↳ ----- lambda_6 (Lambda)	(None, 1)	0	input_1[0][0]
↳ ----- lambda_7 (Lambda)	(None, 1)	0	input_1[0][0]
↳ ----- embedding (Embedding)	(None, 1, 5)	50	lambda[0][0]
↳ ----- embedding_1 (Embedding)	(None, 1, 4)	32	lambda_1[0][0]
↳ ----- embedding_2 (Embedding)	(None, 1, 2)	10	lambda_2[0][0]
↳ ----- embedding_3 (Embedding)	(None, 1, 5)	50	lambda_3[0][0]
↳ ----- embedding_4 (Embedding)	(None, 1, 3)	21	lambda_4[0][0]
↳ ----- embedding_5 (Embedding)	(None, 1, 3)	18	lambda_5[0][0]
↳ ----- embedding_6 (Embedding)	(None, 1, 2)	6	lambda_6[0][0]
↳ ----- embedding_7 (Embedding)	(None, 1, 6)	72	lambda_7[0][0]
↳ -----			

(continues on next page)

(continued from previous page)

lambda_8 (Lambda)	(None, 4)	0	input_1[0][0]
↪ reshape (Reshape)	(None, 5)	0	embedding[0][0]
↪ reshape_1 (Reshape)	(None, 4)	0	embedding_1[0][0]
↪ reshape_2 (Reshape)	(None, 2)	0	embedding_2[0][0]
↪ reshape_3 (Reshape)	(None, 5)	0	embedding_3[0][0]
↪ reshape_4 (Reshape)	(None, 3)	0	embedding_4[0][0]
↪ reshape_5 (Reshape)	(None, 3)	0	embedding_5[0][0]
↪ reshape_6 (Reshape)	(None, 2)	0	embedding_6[0][0]
↪ reshape_7 (Reshape)	(None, 6)	0	embedding_7[0][0]
↪ dense (Dense)	(None, 16)	80	lambda_8[0][0]
↪ concatenate (Concatenate)	(None, 46)	0	reshape[0][0] reshape_1[0][0] reshape_2[0][0] reshape_3[0][0] reshape_4[0][0] reshape_5[0][0] reshape_6[0][0] reshape_7[0][0] dense[0][0]
↪ dense_1 (Dense)	(None, 60)	2820	concatenate[0][0]
↪ dropout (Dropout)	(None, 60)	0	dense_1[0][0]
↪ dense_2 (Dense)	(None, 60)	3660	dropout[0][0]
↪ dropout_1 (Dropout)	(None, 60)	0	dense_2[0][0]

(continues on next page)

(continued from previous page)

```

↪-----
dense_3 (Dense)                (None, 60)                3660                dropout_1[0][0]
-----
↪-----
dropout_2 (Dropout)            (None, 60)                 0                  dense_3[0][0]
-----
↪-----
dense_4 (Dense)                (None, 2)                 122                dropout_2[0][0]
=====
Total params: 10,601
Trainable params: 10,601
Non-trainable params: 0
-----
↪-----

```

```
[13]: <tensorflow.python.keras.callbacks.History at 0x7f482905f8d0>
```

Generate counterfactual

Original instance:

```
[14]: X = X_test[0].reshape((1,) + X_test[0].shape)
```

Initialize counterfactual parameters:

```
[15]: shape = X.shape
      beta = .01
      c_init = 1.
      c_steps = 5
      max_iterations = 500
      rng = (-1., 1.) # scale features between -1 and 1
      rng_shape = (1,) + data.shape[1:]
      feature_range = ((np.ones(rng_shape) * rng[0]).astype(np.float32),
                       (np.ones(rng_shape) * rng[1]).astype(np.float32))
```

Initialize explainer. Since the Embedding layers in `tf.keras` do not let gradients propagate through, we will only make use of the model's `predict` function, treat it as a black box and perform numerical gradient calculations.

```
[16]: set_seed()

# define predict function
predict_fn = lambda x: nn.predict(x)

cf = CounterfactualProto(predict_fn,
                          shape,
                          beta=beta,
                          cat_vars=cat_vars_ord,
                          max_iterations=max_iterations,
                          feature_range=feature_range,
                          c_init=c_init,
                          c_steps=c_steps,
```

(continues on next page)

(continued from previous page)

```

        eps=(.01, .01) # perturbation size for numerical gradients
    )

```

Fit explainer. Please check the [documentation](#) for more info about the optional arguments.

```
[17]: cf.fit(X_train, d_type='abdm', disc_perc=[25, 50, 75]);
```

Explain instance:

```
[18]: set_seed()
      explanation = cf.explain(X)
```

Helper function to more clearly describe explanations:

```
[19]: def describe_instance(X, explanation, eps=1e-2):
      print('Original instance: {} -- proba: {}'.format(target_names[explanation.orig_
      ↪class],
                                                         explanation.orig_proba[0]))
      print('Counterfactual instance: {} -- proba: {}'.format(target_names[explanation.cf[
      ↪'class']],
                                                         explanation.cf['proba'][0]))

      print('\nCounterfactual perturbations...')
      print('\nCategorical:')
      X_orig_ord = X
      X_cf_ord = explanation.cf['X']
      delta_cat = {}
      for i, (_, v) in enumerate(category_map.items()):
          cat_orig = v[int(X_orig_ord[0, i])]
          cat_cf = v[int(X_cf_ord[0, i])]
          if cat_orig != cat_cf:
              delta_cat[feature_names[i]] = [cat_orig, cat_cf]
      if delta_cat:
          for k, v in delta_cat.items():
              print('{}: {} --> {}'.format(k, v[0], v[1]))
      print('\nNumerical:')
      delta_num = X_cf_ord[0, -4:] - X_orig_ord[0, -4:]
      n_keys = len(list(cat_vars_ord.keys()))
      for i in range(delta_num.shape[0]):
          if np.abs(delta_num[i]) > eps:
              print('{}: {:.2f} --> {:.2f}'.format(feature_names[i+n_keys],
                                                         X_orig_ord[0, i+n_keys],
                                                         X_cf_ord[0, i+n_keys]))

```

```
[20]: describe_instance(X, explanation)
```

```

Original instance: <=50K -- proba: [0.6976237  0.30237624]
Counterfactual instance: >50K -- proba: [0.49604183 0.5039582 ]

```

Counterfactual perturbations...

Categorical:

(continues on next page)

(continued from previous page)

```
Numerical:
Capital Gain: -1.00 --> -0.88
```

The person's income is predicted to be above \$50k by increasing his or her capital gain.

8.6.3 Counterfactuals guided by prototypes on California housing dataset

This notebook goes through an example of *prototypical counterfactuals* using *k-d trees* to build the prototypes. Please check out [this notebook](#) for a more in-depth application of the method on MNIST using (auto-)encoders and trust scores.

In this example, we will train a simple neural net to predict whether house prices in California districts are above the median value or not. We can then find a counterfactual to see which variables need to be changed to increase or decrease a house price above or below the median value.

Note

To enable support for CounterfactualProto, you may need to run

```
pip install alibi[tensorflow]
```

```
[2]: %matplotlib inline
import matplotlib.pyplot as plt

import tensorflow as tf
tf.get_logger().setLevel(40) # suppress deprecation messages
tf.compat.v1.disable_v2_behavior() # disable TF2 behaviour as alibi code still relies on_
↳ TF1 constructs
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.utils import to_categorical

import os
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from alibi.explainers import CounterfactualProto

print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False

TF version: 2.7.4
Eager execution enabled: False
```

Load and prepare California housing dataset

```
[3]: california = fetch_california_housing(as_frame=True)
X = california.data.to_numpy()
target = california.target.to_numpy()
feature_names = california.feature_names
```

```
[4]: california.data.head()
```

```
[4]:   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0   8.3252    41.0    6.984127   1.023810     322.0    2.555556    37.88
1   8.3014    21.0    6.238137   0.971880    2401.0    2.109842    37.86
2   7.2574    52.0    8.288136   1.073446     496.0    2.802260    37.85
3   5.6431    52.0    5.817352   1.073059     558.0    2.547945    37.85
4   3.8462    52.0    6.281853   1.081081     565.0    2.181467    37.85

   Longitude
0   -122.23
1   -122.22
2   -122.24
3   -122.25
4   -122.25
```

Each row represents a whole census group. Explanation of features:

- MedInc - median income in block group
- HouseAge - median house age in block group
- AveRooms - average number of rooms per household
- AveBedrms - average number of bedrooms per household
- Population - block group population
- AveOccup - average number of household members
- Latitude - block group latitude
- Longitude - block group longitude

For more details on the dataset, refer to the [scikit-learn documentation](#).

Transform into classification task: target becomes whether house price is above the overall median or not

```
[5]: y = np.zeros((target.shape[0],))
y[np.where(target > np.median(target))[0]] = 1
```

Standardize data

```
[6]: mu = X.mean(axis=0)
sigma = X.std(axis=0)
X = (X - mu) / sigma
```

Define train and test set

```
[7]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Train model

```
[8]: np.random.seed(42)
     tf.random.set_seed(42)
```

```
[9]: def nn_model():
     x_in = Input(shape=(8,))
     x = Dense(40, activation='relu')(x_in)
     x = Dense(40, activation='relu')(x)
     x_out = Dense(2, activation='softmax')(x)
     nn = Model(inputs=x_in, outputs=x_out)
     nn.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
     return nn
```

```
[ ]: nn = nn_model()
     nn.summary()
     nn.fit(X_train, y_train, batch_size=64, epochs=500, verbose=0)
     nn.save('nn_california.h5', save_format='h5')
```

```
[12]: nn = load_model('nn_california.h5')
     score = nn.evaluate(X_test, y_test, verbose=0)
     print('Test accuracy: ', score[1])
```

```
Test accuracy:  0.87863374
```

Generate counterfactual guided by the nearest class prototype

Original instance:

```
[13]: X = X_test[1].reshape((1,) + X_test[1].shape)
     shape = X.shape
```

Run counterfactual:

```
[ ]: # define model
     nn = load_model('nn_california.h5')

     # initialize and fit the explainer
     cf = CounterfactualProto(nn, shape, use_kdtree=True, theta=10., max_iterations=1000,
                             feature_range=(X_train.min(axis=0), X_train.max(axis=0)),
                             c_init=1., c_steps=10)

     cf.fit(X_train)
```

```
[18]: # generate a counterfactual
     explanation = cf.explain(X)
```

The prediction flipped from 0 (value below the median) to 1 (above the median):

```
[20]: print(f'Original prediction: {explanation.orig_class}')
     print(f'Counterfactual prediction: {explanation.cf["class"]}')

```

```
Original prediction: 0
Counterfactual prediction: 1
```

Let's take a look at the counterfactual. To make the results more interpretable, we will first undo the pre-processing step and then check where the counterfactual differs from the original instance:

```
[21]: orig = X * sigma + mu
      counterfactual = explanation.cf['X'] * sigma + mu
      delta = counterfactual - orig
      for i, f in enumerate(feature_names):
          if np.abs(delta[0][i]) > 1e-4:
              print(f'{f}: {delta[0][i]}')
```

```
AveOccup: -0.9049749915631999
Latitude: -0.31885583625280134
```

So in order for the model to consider the census group as having above median house prices, the average occupancy would have to be lower by almost a whole household member, and the location of the census group would need to shift slightly South.

Comparing the original instance and the counterfactual side-by-side:

```
[22]: pd.DataFrame(orig, columns=feature_names)
```

```
[22]:   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  2.5313      30.0   5.039384   1.193493    1565.0    2.679795     35.14

      Longitude
0    -119.46
```

```
[23]: pd.DataFrame(counterfactual, columns=feature_names)
```

```
[23]:   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  2.5313      30.0   5.039384   1.193493    1565.0000004    1.77482    34.821144

      Longitude
0    -119.46
```

Clean up:

```
[24]: os.remove('nn_california.h5')
```

8.6.4 Counterfactuals guided by prototypes on MNIST

This method is described in the [Interpretable Counterfactual Explanations Guided by Prototypes](#) paper and can generate counterfactual instances guided by class prototypes. It means that for a certain instance X , the method builds a prototype for each prediction class using either an [autoencoder](#) or [k-d trees](#). The nearest prototype class other than the originally predicted class is then used to guide the counterfactual search. For example, in MNIST the closest class to a 7 could be a 9. As a result, the prototype loss term will try to minimize the distance between the proposed counterfactual and the prototype of a 9. This speeds up the search towards a satisfactory counterfactual by steering it towards an interpretable solution from the start of the optimization. It also helps to avoid out-of-distribution counterfactuals with the perturbations driven to a prototype of another class.

The loss function to be optimized is the following:

$$Loss = cL_{pred} + \beta L_1 + L_2 + L_{AE} + L_{proto}$$

The first loss term relates to the model's prediction function, the following 2 terms define the elastic net regularization while the last 2 terms are optional. The aim of L_{AE} is to penalize out-of-distribution counterfactuals while L_{proto} guides the counterfactual to a prototype. When we only have access to the model's prediction function and cannot fully enjoy the benefits of automatic differentiation, the prototypes allow us to drop the prediction function loss term L_{pred} and still generate high quality counterfactuals. This drastically reduces the number of prediction calls made during the numerical gradient update step and again speeds up the search.

Other options include generating counterfactuals for specific classes or including trust score constraints to ensure that the counterfactual is close enough to the newly predicted class compared to the original class. Different use cases are illustrated throughout this notebook.

Note

To enable support for CounterfactualProto, you may need to run

```
pip install alibi[tensorflow]
```

```
[1]: import tensorflow as tf
tf.get_logger().setLevel(40) # suppress deprecation messages
tf.compat.v1.disable_v2_behavior() # disable TF2 behaviour as alibi code still relies on_
↳ TF1 constructs
from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input,
↳ UpSampling2D
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.utils import to_categorical

import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
from time import time
from alibi.explainers import CounterfactualProto

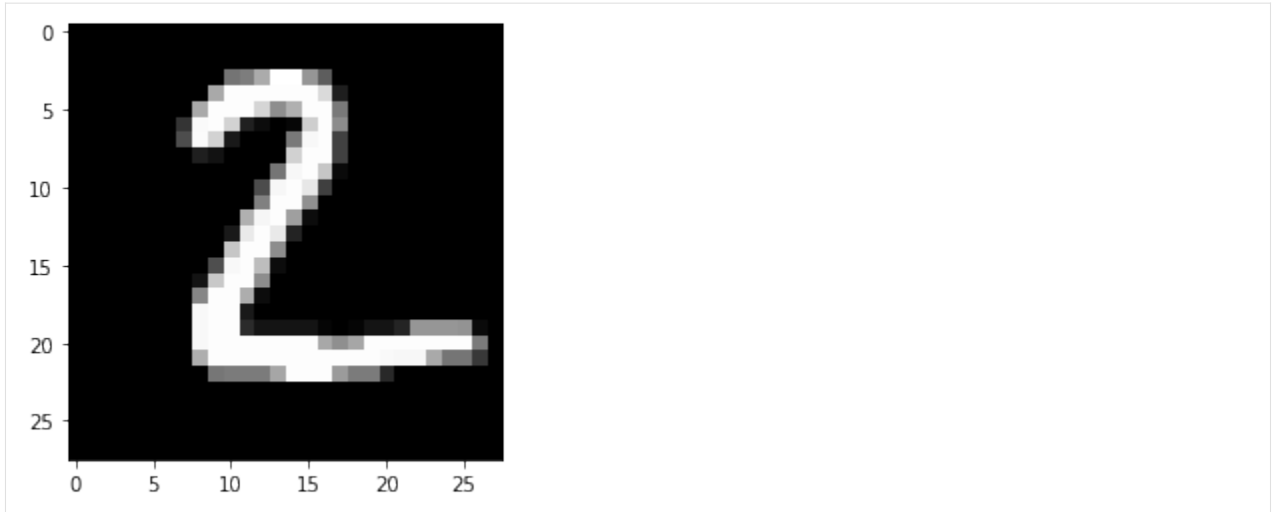
print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # False
```

```
TF version: 2.2.0
Eager execution enabled: False
```

Load and prepare MNIST data

```
[2]: (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)
plt.gray()
plt.imshow(x_test[1]);

x_train shape: (60000, 28, 28) y_train shape: (60000,)
```

Prepare data: scale, reshape and categorize

```
[3]: x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = np.reshape(x_train, x_train.shape + (1,))
x_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)

x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
y_train shape: (60000, 10) y_test shape: (10000, 10)
```

```
[4]: xmin, xmax = -.5, .5
x_train = ((x_train - x_train.min()) / (x_train.max() - x_train.min())) * (xmax - xmin) + xmin
x_test = ((x_test - x_test.min()) / (x_test.max() - x_test.min())) * (xmax - xmin) + xmin
```

Define and train CNN model

```
[5]: def cnn_model():
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')(x_in)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)

    x = Conv2D(filters=64, kernel_size=2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(0.3)(x)

    x = Flatten()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.5)(x)
    x_out = Dense(10, activation='softmax')(x)
```

(continues on next page)

(continued from previous page)

```

cnn = Model(inputs=x_in, outputs=x_out)
cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

return cnn

```

```

[6]: cnn = cnn_model()
cnn.fit(x_train, y_train, batch_size=32, epochs=3, verbose=0)
cnn.save('mnist_cnn.h5', save_format='h5')

```

Evaluate the model on test set

```

[7]: cnn = load_model('mnist_cnn.h5')
score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])

```

Test accuracy: 0.9871

Define and train auto-encoder

```

[8]: def ae_model():
    # encoder
    x_in = Input(shape=(28, 28, 1))
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x_in)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)
    encoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)
    encoder = Model(x_in, encoded)

    # decoder
    dec_in = Input(shape=(14, 14, 1))
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(dec_in)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    decoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)
    decoder = Model(dec_in, decoded)

    # autoencoder = encoder + decoder
    x_out = decoder(encoder(x_in))
    autoencoder = Model(x_in, x_out)
    autoencoder.compile(optimizer='adam', loss='mse')

    return autoencoder, encoder, decoder

```

```

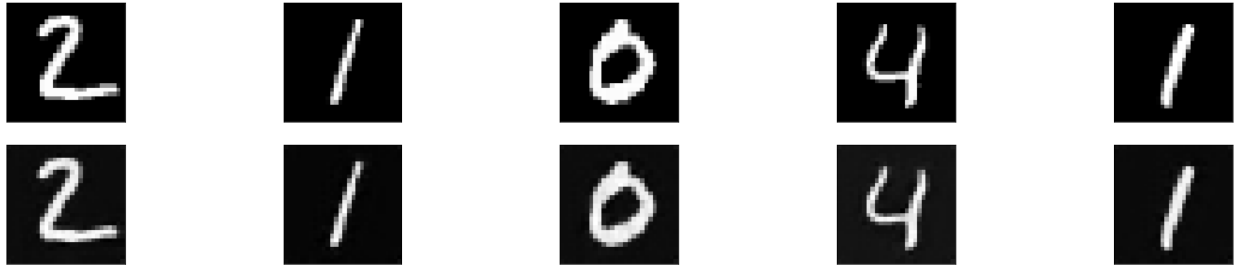
[9]: ae, enc, dec = ae_model()
ae.fit(x_train, x_train, batch_size=128, epochs=4, validation_data=(x_test, x_test),
      verbose=0)
ae.save('mnist_ae.h5', save_format='h5')
enc.save('mnist_enc.h5', save_format='h5')

```

Compare original with decoded images

```
[10]: ae = load_model('mnist_ae.h5')
      enc = load_model('mnist_enc.h5', compile=False)

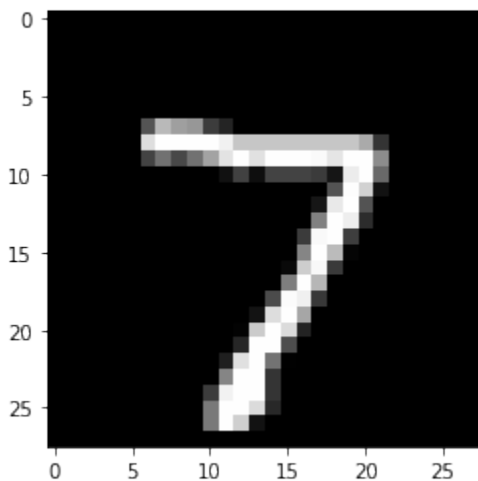
      decoded_imgs = ae.predict(x_test)
      n = 5
      plt.figure(figsize=(20, 4))
      for i in range(1, n+1):
          # display original
          ax = plt.subplot(2, n, i)
          plt.imshow(x_test[i].reshape(28, 28))
          ax.get_xaxis().set_visible(False)
          ax.get_yaxis().set_visible(False)
          # display reconstruction
          ax = plt.subplot(2, n, i + n)
          plt.imshow(decoded_imgs[i].reshape(28, 28))
          ax.get_xaxis().set_visible(False)
          ax.get_yaxis().set_visible(False)
      plt.show()
```



Generate counterfactual guided by the nearest class prototype

Original instance:

```
[11]: X = x_test[0].reshape((1,) + x_test[0].shape)
      plt.imshow(X.reshape(28, 28));
```



Counterfactual parameters:

```
[12]: shape = (1,) + x_train.shape[1:]
      gamma = 100.
      theta = 100.
      c_init = 1.
      c_steps = 2
      max_iterations = 1000
      feature_range = (x_train.min(), x_train.max())
```

Run counterfactual:

```
[13]: # initialize explainer, fit and generate counterfactual
      cf = CounterfactualProto(cnn, shape, gamma=gamma, theta=theta,
                              ae_model=ae, enc_model=enc, max_iterations=max_iterations,
                              feature_range=feature_range, c_init=c_init, c_steps=c_steps)

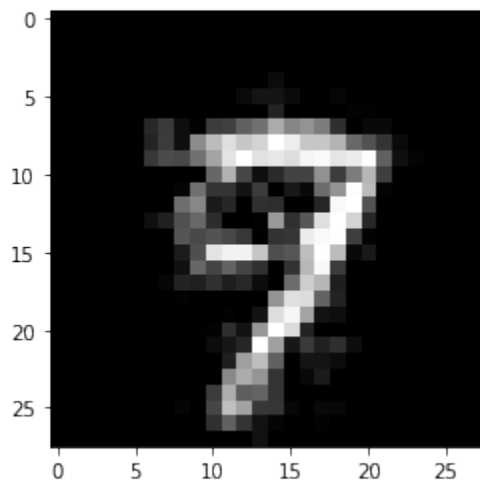
      start_time = time()
      cf.fit(x_train) # find class prototypes
      print('Time to find prototypes each class: {:.3f} sec'.format(time() - start_time))
      start_time = time()
      explanation = cf.explain(X)
      print('Explanation took {:.3f} sec'.format(time() - start_time))

      Time to find prototypes each class: 14.580 sec
      Explanation took 9.269 sec
```

Results:

```
[14]: print('Counterfactual prediction: {}'.format(explanation.cf['class']))
      print(f'Closest prototype class: {explanation.id_proto}')
      plt.imshow(explanation.cf['X'].reshape(28, 28));
```

Counterfactual prediction: 9
Closest prototype class: 9



The counterfactual starting from a 7 moves towards its closest prototype class: a 9. The evolution of the counterfactual during the first iteration can be seen below:

```
[15]: iter_cf = 0
      print(f'iteration c {iter_cf}')
```

(continues on next page)

(continued from previous page)

```

n = len(explanation['all'][iter_cf])
plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(1, n+1, i+1)
    plt.imshow(explanation['all'][iter_cf][i].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()

```

iteration c 0



Typically, the first few iterations already steer the 7 towards a 9, while the later iterations make the counterfactual more sparse.

Prototypes defined by the k nearest encoded instances

In the above example, the class prototypes are defined by the average encoding of all instances belonging to the specific class. Instead, we can also select only the k nearest encoded instances of a class to the encoded instance to be explained and use the average over those k encodings as the prototype.

```

[16]: # initialize explainer, fit and generate counterfactuals
cf = CounterfactualProto(cnn, shape, gamma=gamma, theta=theta,
                        ae_model=ae, enc_model=enc, max_iterations=max_iterations,
                        feature_range=feature_range, c_init=c_init, c_steps=c_steps)

cf.fit(x_train)
explanation_k1 = cf.explain(X, k=1, k_type='mean')
explanation_k20 = cf.explain(X, k=20, k_type='mean')

```

Results for k equals 1:

```

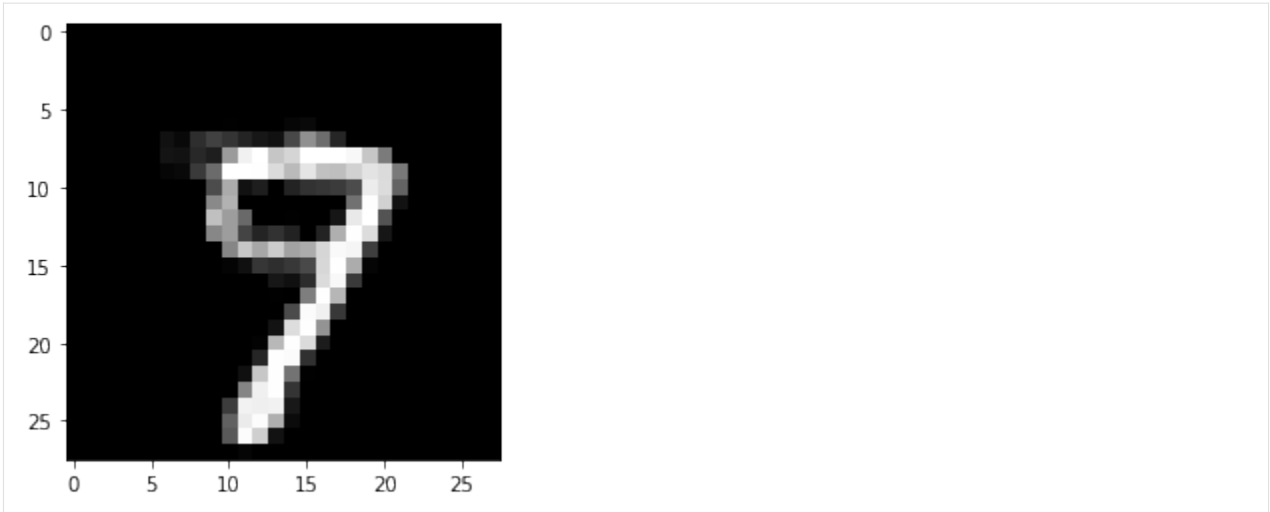
[17]: print('Counterfactual prediction: {}'.format(explanation_k1.cf['class']))
print(f'Closest prototype class: {explanation_k1.id_proto}')
plt.imshow(explanation_k1.cf['X'].reshape(28, 28));

```

```

Counterfactual prediction: 9
Closest prototype class: 9

```

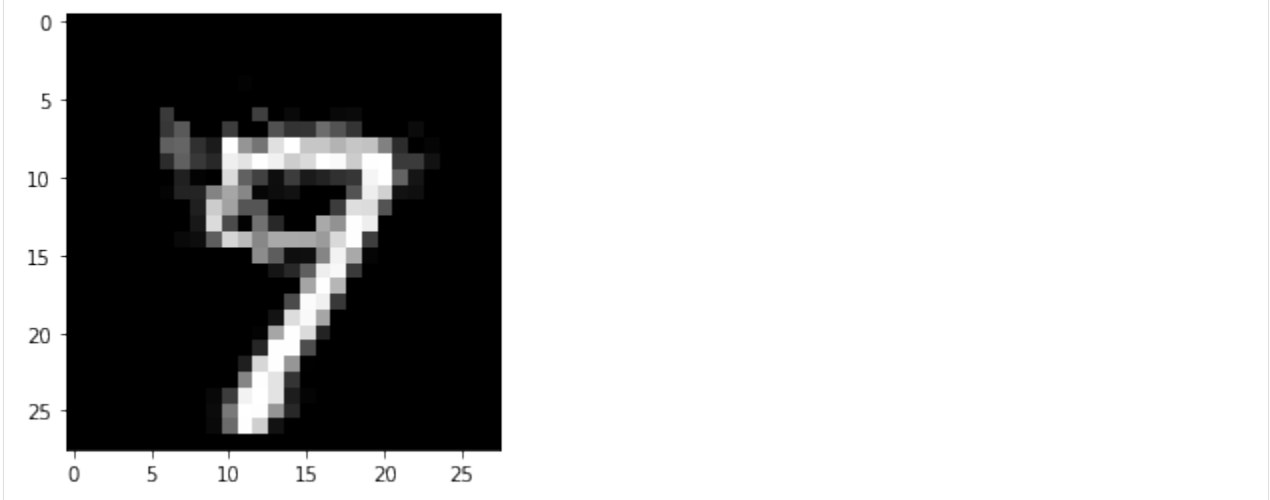


Results for k equals 20:

```
[18]: print('Counterfactual prediction: {}'.format(explanation_k20.cf['class']))  
      print(f'Closest prototype class: {explanation.id_proto}')  
      plt.imshow(explanation_k20.cf['X'].reshape(28, 28));
```

Counterfactual prediction: 9

Closest prototype class: 9



A lower value of k typically leads to counterfactuals that look more like the original instance and less like an average instance of the counterfactual class.

Remove the autoencoder loss term L_{AE}

In the previous example, we used both an autoencoder loss term to penalize a counterfactual which falls outside of the training data distribution as well as an encoder loss term to guide the counterfactual to the nearest prototype class. In the next example we get rid of the autoencoder loss term to speed up the counterfactual search and still generate decent counterfactuals:

```
[19]: # initialize explainer, fit and generate counterfactuals
cf = CounterfactualProto(cnn, shape, gamma=gamma, theta=theta,
                        enc_model=enc, max_iterations=max_iterations,
                        feature_range=feature_range, c_init=c_init, c_steps=c_steps)

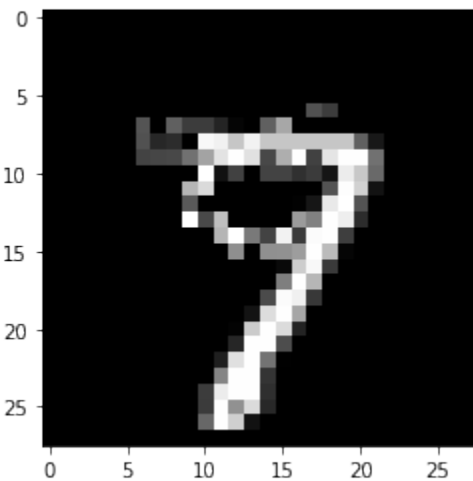
cf.fit(x_train)
start_time = time()
explanation = cf.explain(X, k=1)
print('Explanation took {:.3f} sec'.format(time() - start_time))

Explanation took 6.443 sec
```

Results:

```
[20]: print('Counterfactual prediction: {}'.format(explanation.cf['class']))
print(f'Closest prototype class: {explanation.id_proto}')
plt.imshow(explanation.cf['X'].reshape(28, 28));
```

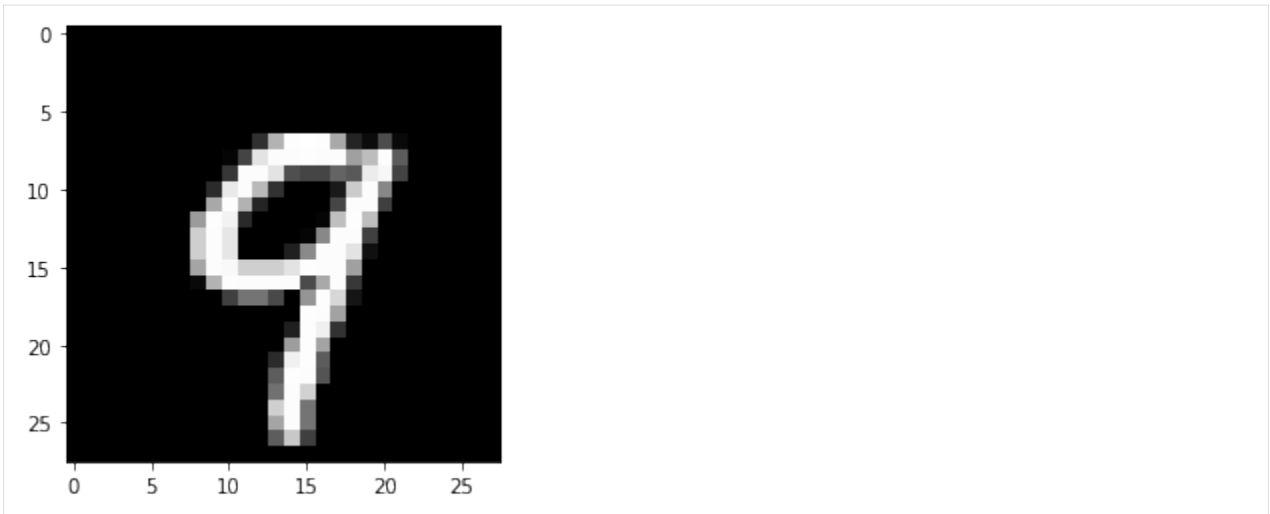
```
Counterfactual prediction: 9
Closest prototype class: 9
```



Specify prototype classes

For multi-class predictions, we might be interested to generate counterfactuals for certain classes while avoiding others. The following example illustrates how to do this:

```
[21]: X = x_test[12].reshape((1,) + x_test[1].shape)
plt.imshow(X.reshape(28, 28));
```



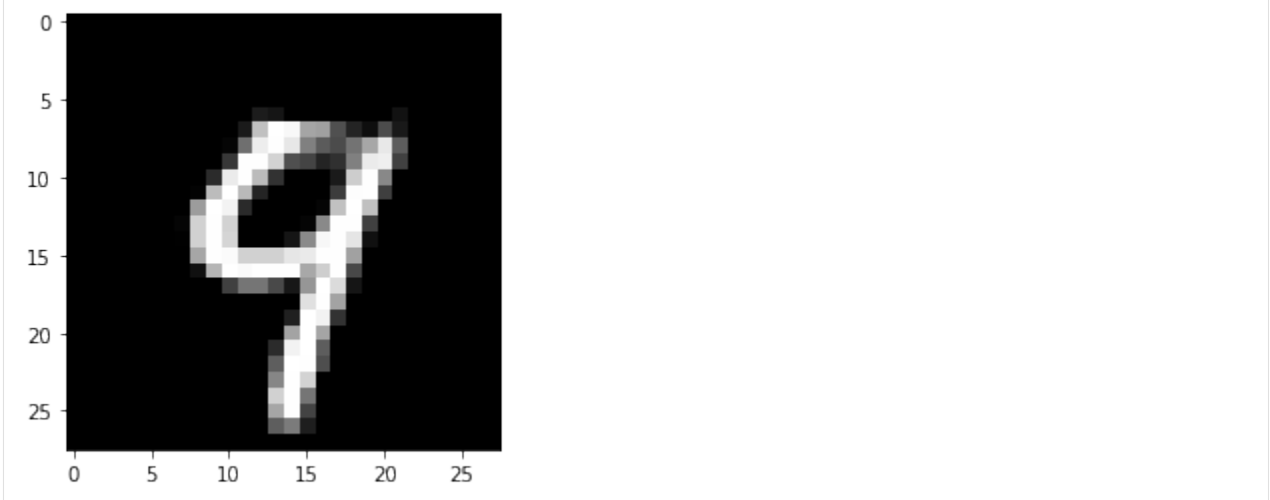
```
[22]: # initialize explainer, fit and generate counterfactuals
cf = CounterfactualProto(cnn, shape, gamma=gamma, theta=theta,
                        ae_model=ae, enc_model=enc, max_iterations=max_iterations,
                        feature_range=feature_range, c_init=c_init, c_steps=c_steps)

cf.fit(x_train)
explanation_1 = cf.explain(X, k=5, k_type='mean')
proto_1 = explanation_1.id_proto
explanation_2 = cf.explain(X, k=5, k_type='mean', target_class=[7])
proto_2 = explanation_2.id_proto
```

The closest class to the 9 is 4. This is evident by looking at the first counterfactual below. For the second counterfactual, we specified that the prototype class used in the search should be a 7. As a result, a counterfactual 7 instead of a 4 is generated.

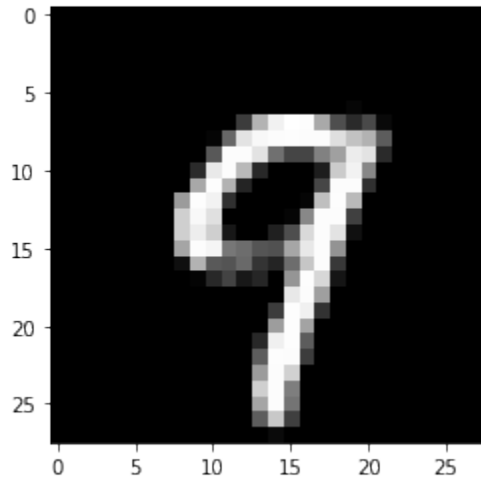
```
[23]: print('Counterfactual prediction: {}'.format(explanation_1.cf['class']))
print(f'Closest prototype class: {proto_1}')
plt.imshow(explanation_1.cf['X'].reshape(28, 28));
```

```
Counterfactual prediction: 4
Closest prototype class: 4
```




```
[24]: print('Counterfactual prediction: {}'.format(explanation_2.cf['class']))
print(f'Closest prototype class: {proto_2}')
plt.imshow(explanation_2.cf['X'].reshape(28, 28));
```

```
Counterfactual prediction: 7
Closest prototype class: 7
```



Speed up the counterfactual search by removing the predict function loss term

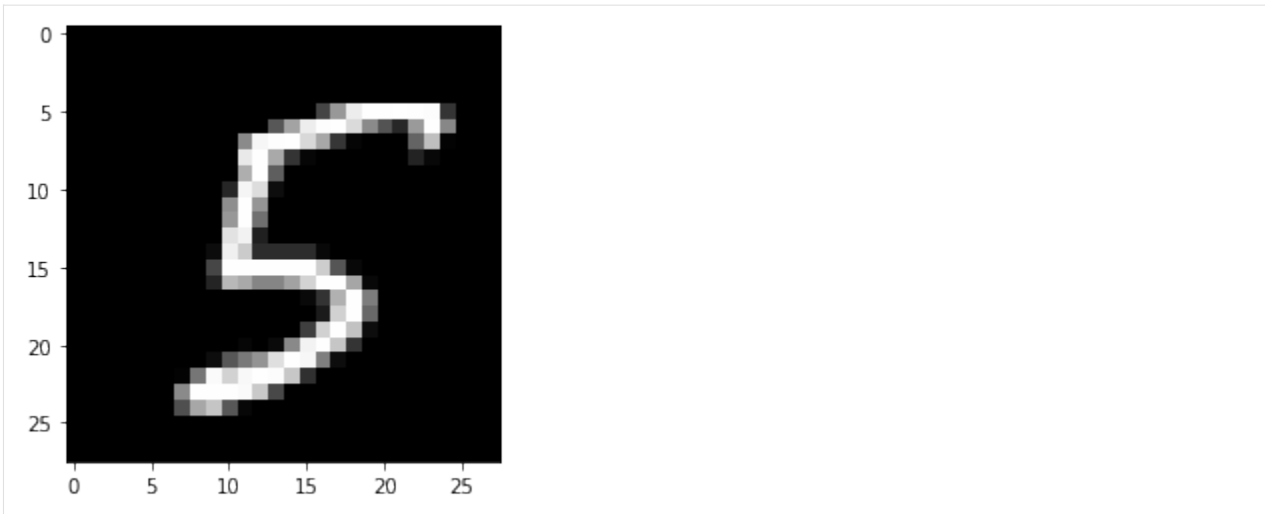
We can also remove the prediction loss term and still obtain an interpretable counterfactual. This is especially relevant for fully black box models. When we provide the counterfactual search method with a Keras or TensorFlow model, it is incorporated in the TensorFlow graph and evaluated using automatic differentiation. However, if we only have access to the model's prediction function, the gradient updates are numerical and typically require a large number of prediction calls because of the prediction loss term L_{pred} . These prediction calls can slow the search down significantly and become a bottleneck. We can represent the gradient of the loss term as follows:

$$\frac{\partial L_{pred}}{\partial x} = \frac{\partial L_{pred}}{\partial p} \frac{\partial p}{\partial x}$$

where L_{pred} is the prediction loss term, p the prediction function and x the input features to optimize. For a 28 by 28 MNIST image, the $\delta p / \delta x$ term alone would require a prediction call with batch size $28 \times 28 \times 2 = 1568$. By using the prototypes to guide the search however, we can remove the prediction loss term and only make a single prediction at the end of each gradient update to check whether the predicted class on the proposed counterfactual is different from the original class. We do not necessarily need a Keras or TensorFlow auto-encoder either and can use k-d trees to find the nearest class prototypes. Please check out [this notebook](#) for a practical example.

The first example below removes L_{pred} from the loss function to bypass the bottleneck. It illustrates the drastic speed improvements over the black box alternative with numerical gradient evaluation while still producing interpretable counterfactual instances.

```
[25]: plt.gray()
X = x_test[23].reshape(1, 28, 28, 1)
plt.imshow(X.reshape(28, 28));
```



```
[26]: c_init = 0. # weight on prediction loss term set to 0
      c_steps = 1 # no need to find optimal values for c
```

```
[27]: # define a black-box model
      predict_fn = lambda x: cnn.predict(x)

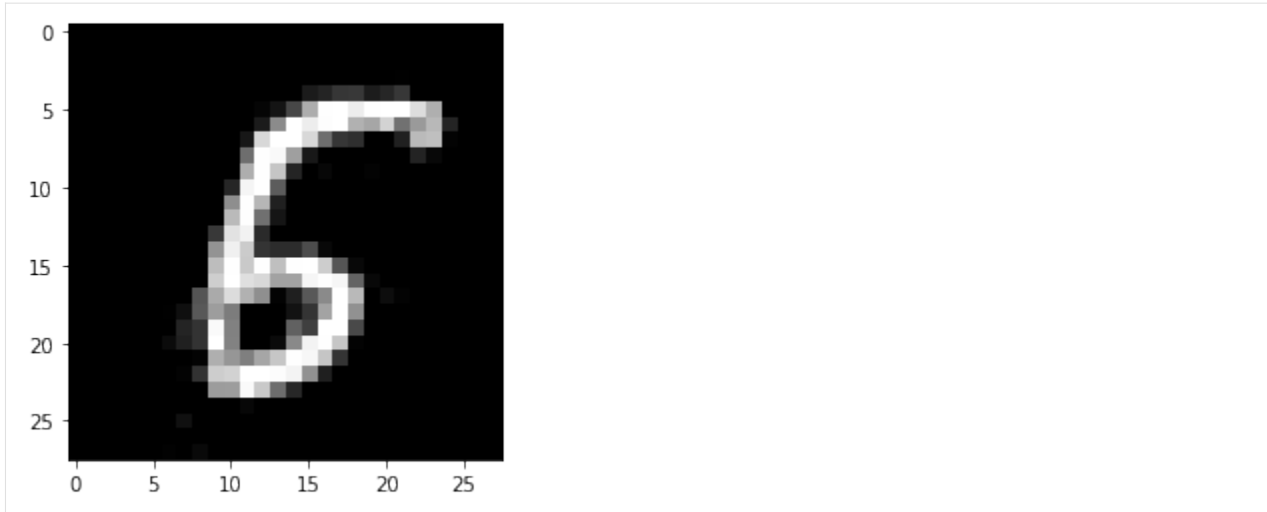
      # initialize explainer, fit and generate counterfactuals
      cf = CounterfactualProto(predict_fn, shape, gamma=gamma, theta=theta,
                              ae_model=ae, enc_model=enc, max_iterations=max_iterations,
                              feature_range=feature_range, c_init=c_init, c_steps=c_steps)

      cf.fit(x_train)
      start_time = time()
      explanation = cf.explain(X, k=1)
      print('Explanation took {:.3f} sec'.format(time() - start_time))

      Explanation took 7.257 sec
```

```
[28]: print('Counterfactual prediction: {}'.format(explanation.cf['class']))
      print(f'Closest prototype class: {explanation.id_proto}')
      plt.imshow(explanation.cf['X'].reshape(28, 28));

      Counterfactual prediction: 6
      Closest prototype class: 6
```

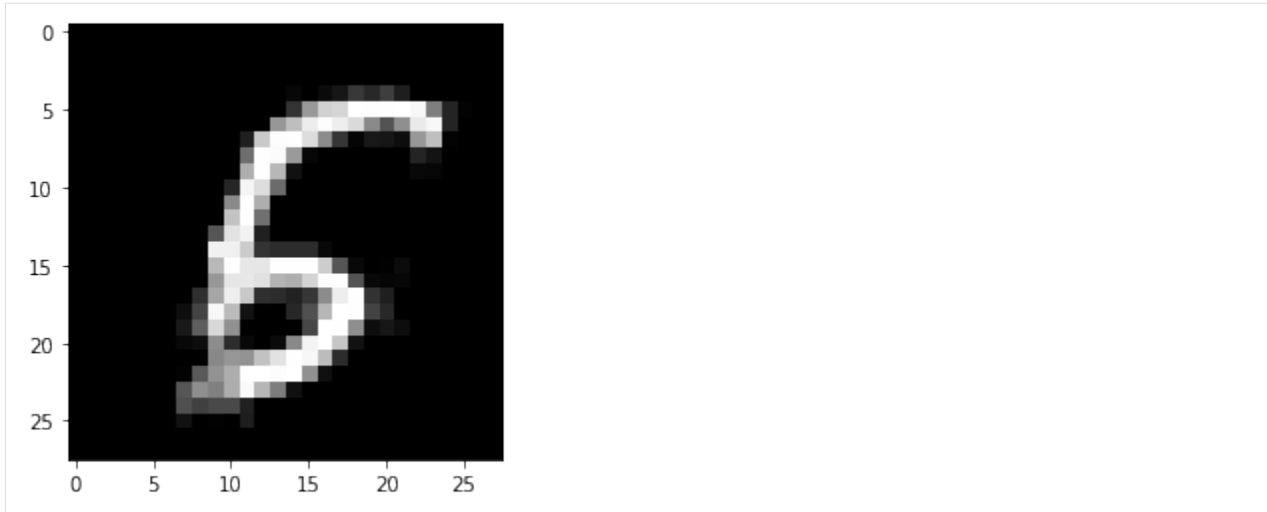


Let us now add the L_{pred} loss term back in the objective function and observe how long it takes to generate a black box counterfactual:

```
[29]: c_init = 1.  
      c_steps = 2
```

```
[30]: # define a black-box model  
      predict_fn = lambda x: cnn.predict(x)  
  
      # initialize explainer, fit and generate counterfactuals  
      cf = CounterfactualProto(predict_fn, shape, gamma=gamma, theta=theta,  
                               ae_model=ae, enc_model=enc, max_iterations=max_iterations,  
                               feature_range=feature_range, c_init=c_init, c_steps=c_steps)  
  
      cf.fit(x_train)  
      start_time = time()  
      explanation = cf.explain(X, k=1)  
      print('Explanation took {:.3f} sec'.format(time() - start_time))  
  
Explanation took 966.342 sec
```

```
[31]: print('Counterfactual prediction: {}'.format(explanation.cf['class']))  
      print(f'Closest prototype class: {explanation.id_proto}')  
      plt.imshow(explanation.cf['X'].reshape(28, 28));  
  
Counterfactual prediction: 6  
Closest prototype class: 6
```



Clean up:

```
[32]: os.remove('mnist_cnn.h5')
      os.remove('mnist_ae.h5')
      os.remove('mnist_enc.h5')
```

8.7 Counterfactuals with Reinforcement Learning

8.7.1 Counterfactual with Reinforcement Learning (CFRL) on Adult Census

This method is described in [Model-agnostic and Scalable Counterfactual Explanations via Reinforcement Learning](#) and can generate counterfactual instances for any black-box model. The usual optimization procedure is transformed into a learnable process allowing to generate batches of counterfactual instances in a single forward pass even for high dimensional data. The training pipeline is model-agnostic and relies only on prediction feedback by querying the black-box model. Furthermore, the method allows target and feature conditioning.

We exemplify the use case for the TensorFlow backend. This means that all models: the autoencoder, the actor and the critic are TensorFlow models. Our implementation supports PyTorch backend as well.

CFRL uses [Deep Deterministic Policy Gradient \(DDPG\)](#) by interleaving a state-action function approximator called critic, with a learning an approximator called actor to predict the optimal action. The method assumes that the critic is differentiable with respect to the action argument, thus allowing to optimize the actor's parameters efficiently through gradient-based methods.

The DDPG algorithm requires two separate networks, an actor μ and a critic Q . Given the encoded representation of the input instance $z = enc(x)$, the model prediction y_M , the target prediction y_T and the conditioning vector c , the actor outputs the counterfactual's latent representation $z_{CF} = \mu(z, y_M, y_T, c)$. The decoder then projects the embedding z_{CF} back to the original input space, followed by optional post-processing.

The training step consists of simultaneously optimizing the actor and critic networks. The critic regresses on the reward R determined by the model prediction, while the actor maximizes the critic's output for the given instance through L_{max} . The actor also minimizes two objectives to encourage the generation of sparse, in-distribution counterfactuals. The sparsity loss $L_{sparsity}$ operates on the decoded counterfactual x_{CF} and combines the L_1 loss over the standardized numerical features and the L_0 loss over the categorical ones. The consistency loss $L_{consist}$ aims to encode the counterfactual x_{CF} back to the same latent representation where it was decoded from and helps to produce in-distribution counterfactual instances. Formally, the actor's loss can be written as: $L_{actor} = L_{max} + \lambda_1 L_{sparsity} + \lambda_2 L_{consistency}$

This example will use the `xgboost` library, which can be installed with:

Note

To enable support for CounterfactualRLTabular with tensorflow backend, you may need to run

```
pip install alibi[tensorflow]
```

```
[4]: import os
import numpy as np
import pandas as pd
from copy import deepcopy
from typing import List, Tuple, Dict, Callable

import tensorflow as tf
import tensorflow.keras as keras

from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

from xgboost import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression

from alibi.explainers import CounterfactualRLTabular, CounterfactualRL
from alibi.datasets import fetch_adult
from alibi.models.tensorflow import HeAE
from alibi.models.tensorflow import Actor, Critic
from alibi.models.tensorflow import ADULTEncoder, ADULTDecoder
from alibi.explainers.cfml_base import Callback
from alibi.explainers.backends.cfml_tabular import get_he_preprocessor, get_statistics, \
    get_conditional_vector, apply_category_mapping
```

Load Adult Census Dataset

```
[2]: # Fetch adult dataset
adult = fetch_adult()

# Separate columns in numerical and categorical.
categorical_names = [adult.feature_names[i] for i in adult.category_map.keys()]
categorical_ids = list(adult.category_map.keys())

numerical_names = [name for i, name in enumerate(adult.feature_names) if i not in adult.
    ↪category_map.keys()]
numerical_ids = [i for i in range(len(adult.feature_names)) if i not in adult.category_
    ↪map.keys()]
```

(continues on next page)

(continued from previous page)

```
# Split data into train and test
X, Y = adult.data, adult.target
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=13)
```

Train black-box classifier

```
[3]: # Define numerical standard scaler.
num_transf = StandardScaler()

# Define categorical one-hot encoder.
cat_transf = OneHotEncoder(
    categories=[range(len(x)) for x in adult.category_map.values()],
    handle_unknown="ignore"
)

# Define column transformer
preprocessor = ColumnTransformer(
    transformers=[
        ("cat", cat_transf, categorical_ids),
        ("num", num_transf, numerical_ids),
    ],
    sparse_threshold=0
)

[4]: # Fit preprocessor.
preprocessor.fit(X_train)

# Preprocess train and test dataset.
X_train_ohe = preprocessor.transform(X_train)
X_test_ohe = preprocessor.transform(X_test)

[5]: # Select one of the below classifiers.
# clf = XGBClassifier(min_child_weight=0.5, max_depth=3, gamma=0.2)
# clf = LogisticRegression(C=10)
# clf = DecisionTreeClassifier(max_depth=10, min_samples_split=5)
clf = RandomForestClassifier(max_depth=15, min_samples_split=10, n_estimators=50)

# Fit the classifier.
clf.fit(X_train_ohe, Y_train)

[5]: RandomForestClassifier(max_depth=15, min_samples_split=10, n_estimators=50)
```

Define the predictor (black-box)

Now that we've trained the classifier, we can define the black-box model. Note that the output of the black-box is a distribution which can be either a soft-label distribution (probabilities/logits for each class) or a hard-label distribution (one-hot encoding). Internally, CFRL takes the `argmax`. Moreover the output **DOES NOT HAVE TO BE DIFFERENTIABLE**.

```
[6]: # Define prediction function.
predictor = lambda x: clf.predict_proba(preprocessor.transform(x))

[7]: # Compute accuracy.
acc = accuracy_score(y_true=Y_test, y_pred=predictor(X_test).argmax(axis=1))
print("Accuracy: %.3f" % acc)

Accuracy: 0.864
```

Define and train autoencoder

Instead of directly modelling the perturbation vector in the potentially high-dimensional input space, we first train an autoencoder. The weights of the encoder are frozen and the actor applies the counterfactual perturbations in the latent space of the encoder. The pre-trained decoder maps the counterfactual embedding back to the input feature space.

The autoencoder follows a standard design. The model is composed from two submodules, the encoder and the decoder. The forward pass consists of passing the input to the encoder, obtain the input embedding and pass the embedding through the decoder.

```
class HeAE(keras.Model):
    def __init__(self, encoder: keras.Model, decoder: keras.Model, **kwargs) -> None:
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def call(self, x: tf.Tensor, **kwargs):
        z = self.encoder(x)
        x_hat = self.decoder(z)
        return x_hat
```

The heterogeneous variant used in this example uses an additional type checking to ensure that the output of the decoder is a list of tensors.

Heterogeneous dataset require special treatment. In this work we modeled the numerical features by normal distributions with constant standard deviation and categorical features by categorical distributions. Due to the choice of feature modeling, some numerical features can end up having different types than the original numerical features. For example, a feature like Age having the type of `int` can become a `float` due to the autoencoder reconstruction (e.g., Age=26 -> Age=26.3). This behavior can be undesirable. Thus we performed casting when process the output of the autoencoder (decoder component).

```
[8]: # Define attribute types, required for datatype conversion.
feature_types = {"Age": int, "Capital Gain": int, "Capital Loss": int, "Hours per week":
    ->int}

# Define data preprocessor and inverse preprocessor. The invers preprocessor include
    ->datatype conversions.
heae_preprocessor, heae_inv_preprocessor = get_he_preprocessor(X=X_train,
```

(continues on next page)

(continued from previous page)

```

↪ feature_names,
↪ category_map,
↪ types)

feature_names=adult.
category_map=adult.
feature_types=feature_

# Define trainset
trainset_input = heae_preprocessor(X_train).astype(np.float32)
trainset_outputs = {
    "output_1": trainset_input[:, :len(numerical_ids)]
}

for i, cat_id in enumerate(categorical_ids):
    trainset_outputs.update({
        f"output_{i+2}": X_train[:, cat_id]
    })

trainset = tf.data.Dataset.from_tensor_slices((trainset_input, trainset_outputs))
trainset = trainset.shuffle(1024).batch(128, drop_remainder=True)

```

```

[9]: # Define autoencoder path and create dir if it doesn't exist.
heae_path = os.path.join("tensorflow", "ADULT_autoencoder")
if not os.path.exists(heae_path):
    os.makedirs(heae_path)

# Define constants.
EPOCHS = 50          # epochs to train the autoencoder
HIDDEN_DIM = 128     # hidden dimension of the autoencoder
LATENT_DIM = 15      # define latent dimension

# Define output dimensions.
OUTPUT_DIMS = [len(numerical_ids)]
OUTPUT_DIMS += [len(adult.category_map[cat_id]) for cat_id in categorical_ids]

# Define the heterogeneous auto-encoder.
heae = HeAE(encoder=ADULTEncoder(hidden_dim=HIDDEN_DIM, latent_dim=LATENT_DIM),
            decoder=ADULTDecoder(hidden_dim=HIDDEN_DIM, output_dims=OUTPUT_DIMS))

# Define loss functions.
he_loss = [keras.losses.MeanSquaredError()]
he_loss_weights = [1.]

# Add categorical losses.
for i in range(len(categorical_names)):
    he_loss.append(keras.losses.SparseCategoricalCrossentropy(from_logits=True))
    he_loss_weights.append(1./len(categorical_names))

# Define metrics.
metrics = {}
for i, cat_name in enumerate(categorical_names):
    metrics.update({f"output_{i+2}": keras.metrics.SparseCategoricalAccuracy()})

```

(continues on next page)

(continued from previous page)

```
# Compile model.
heae.compile(optimizer=keras.optimizers.Adam(learning_rate=1e-3),
             loss=he_loss,
             loss_weights=he_loss_weights,
             metrics=metrics)

if len(os.listdir(heae_path)) == 0:
    # Fit and save autoencoder.
    heae.fit(trainset, epochs=EPOCHS)
    heae.save(heae_path, save_format="tf")
else:
    # Load the model.
    heae = keras.models.load_model(heae_path, compile=False)
```

Epoch 1/50

```
203/203 [=====] - 3s 6ms/step - loss: 1.1228 - output_1_loss: 0.
↳ 2364 - output_2_loss: 1.1212 - output_3_loss: 1.2091 - output_4_loss: 0.6083 - output_
↳ 5_loss: 1.5602 - output_6_loss: 0.9074 - output_7_loss: 0.6149 - output_8_loss: 0.3439
↳ - output_9_loss: 0.7265 - output_2_sparse_categorical_accuracy: 0.6879 - output_3_
↳ sparse_categorical_accuracy: 0.5755 - output_4_sparse_categorical_accuracy: 0.7886 -
↳ output_5_sparse_categorical_accuracy: 0.4560 - output_6_sparse_categorical_accuracy: 0.
↳ 7181 - output_7_sparse_categorical_accuracy: 0.8123 - output_8_sparse_categorical_
↳ accuracy: 0.8518 - output_9_sparse_categorical_accuracy: 0.8578
```

Epoch 2/50

```
203/203 [=====] - 1s 6ms/step - loss: 0.4056 - output_1_loss: 0.
↳ 0395 - output_2_loss: 0.6040 - output_3_loss: 0.5136 - output_4_loss: 0.1736 - output_
↳ 5_loss: 0.5957 - output_6_loss: 0.3023 - output_7_loss: 0.3132 - output_8_loss: 0.0976
↳ - output_9_loss: 0.3288 - output_2_sparse_categorical_accuracy: 0.7967 - output_3_
↳ sparse_categorical_accuracy: 0.8346 - output_4_sparse_categorical_accuracy: 0.9540 -
↳ output_5_sparse_categorical_accuracy: 0.8339 - output_6_sparse_categorical_accuracy: 0.
↳ 9210 - output_7_sparse_categorical_accuracy: 0.8900 - output_8_sparse_categorical_
↳ accuracy: 0.9735 - output_9_sparse_categorical_accuracy: 0.9092
```

Epoch 3/50

```
203/203 [=====] - 1s 6ms/step - loss: 0.2299 - output_1_loss: 0.
↳ 0288 - output_2_loss: 0.2825 - output_3_loss: 0.2869 - output_4_loss: 0.1087 - output_
↳ 5_loss: 0.2454 - output_6_loss: 0.1982 - output_7_loss: 0.1817 - output_8_loss: 0.0510
↳ - output_9_loss: 0.2541 - output_2_sparse_categorical_accuracy: 0.9271 - output_3_
↳ sparse_categorical_accuracy: 0.9168 - output_4_sparse_categorical_accuracy: 0.9739 -
↳ output_5_sparse_categorical_accuracy: 0.9474 - output_6_sparse_categorical_accuracy: 0.
↳ 9444 - output_7_sparse_categorical_accuracy: 0.9486 - output_8_sparse_categorical_
↳ accuracy: 0.9892 - output_9_sparse_categorical_accuracy: 0.9231
```

Epoch 4/50

```
203/203 [=====] - 1s 6ms/step - loss: 0.1582 - output_1_loss: 0.
↳ 0220 - output_2_loss: 0.1704 - output_3_loss: 0.1952 - output_4_loss: 0.0748 - output_
↳ 5_loss: 0.1633 - output_6_loss: 0.1357 - output_7_loss: 0.1176 - output_8_loss: 0.0324
↳ - output_9_loss: 0.2000 - output_2_sparse_categorical_accuracy: 0.9601 - output_3_
↳ sparse_categorical_accuracy: 0.9452 - output_4_sparse_categorical_accuracy: 0.9812 -
↳ output_5_sparse_categorical_accuracy: 0.9646 - output_6_sparse_categorical_accuracy: 0.
↳ 9643 - output_7_sparse_categorical_accuracy: 0.9671 - output_8_sparse_categorical_
↳ accuracy: 0.9933 - output_9_sparse_categorical_accuracy: 0.9375
```

Epoch 5/50

(continues on next page)

(continued from previous page)

```

203/203 [=====] - 1s 6ms/step - loss: 0.1218 - output_1_loss: 0.
→0175 - output_2_loss: 0.1279 - output_3_loss: 0.1429 - output_4_loss: 0.0577 - output_
→5_loss: 0.1298 - output_6_loss: 0.0969 - output_7_loss: 0.0897 - output_8_loss: 0.0234_
→- output_9_loss: 0.1656 - output_2_sparse_categorical_accuracy: 0.9697 - output_3_
→sparse_categorical_accuracy: 0.9632 - output_4_sparse_categorical_accuracy: 0.9850 -_
→output_5_sparse_categorical_accuracy: 0.9686 - output_6_sparse_categorical_accuracy: 0.
→9757 - output_7_sparse_categorical_accuracy: 0.9726 - output_8_sparse_categorical_
→accuracy: 0.9950 - output_9_sparse_categorical_accuracy: 0.9493

```

Epoch 6/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0986 - output_1_loss: 0.
→0146 - output_2_loss: 0.1042 - output_3_loss: 0.1073 - output_4_loss: 0.0475 - output_
→5_loss: 0.1068 - output_6_loss: 0.0741 - output_7_loss: 0.0734 - output_8_loss: 0.0181_
→- output_9_loss: 0.1410 - output_2_sparse_categorical_accuracy: 0.9757 - output_3_
→sparse_categorical_accuracy: 0.9747 - output_4_sparse_categorical_accuracy: 0.9878 -_
→output_5_sparse_categorical_accuracy: 0.9736 - output_6_sparse_categorical_accuracy: 0.
→9820 - output_7_sparse_categorical_accuracy: 0.9776 - output_8_sparse_categorical_
→accuracy: 0.9959 - output_9_sparse_categorical_accuracy: 0.9582

```

Epoch 7/50

```

203/203 [=====] - 1s 7ms/step - loss: 0.0826 - output_1_loss: 0.
→0126 - output_2_loss: 0.0886 - output_3_loss: 0.0831 - output_4_loss: 0.0408 - output_
→5_loss: 0.0893 - output_6_loss: 0.0592 - output_7_loss: 0.0626 - output_8_loss: 0.0147_
→- output_9_loss: 0.1219 - output_2_sparse_categorical_accuracy: 0.9784 - output_3_
→sparse_categorical_accuracy: 0.9819 - output_4_sparse_categorical_accuracy: 0.9891 -_
→output_5_sparse_categorical_accuracy: 0.9790 - output_6_sparse_categorical_accuracy: 0.
→9856 - output_7_sparse_categorical_accuracy: 0.9816 - output_8_sparse_categorical_
→accuracy: 0.9968 - output_9_sparse_categorical_accuracy: 0.9639

```

Epoch 8/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0713 - output_1_loss: 0.
→0112 - output_2_loss: 0.0771 - output_3_loss: 0.0680 - output_4_loss: 0.0359 - output_
→5_loss: 0.0766 - output_6_loss: 0.0498 - output_7_loss: 0.0533 - output_8_loss: 0.0128_
→- output_9_loss: 0.1069 - output_2_sparse_categorical_accuracy: 0.9808 - output_3_
→sparse_categorical_accuracy: 0.9844 - output_4_sparse_categorical_accuracy: 0.9905 -_
→output_5_sparse_categorical_accuracy: 0.9819 - output_6_sparse_categorical_accuracy: 0.
→9880 - output_7_sparse_categorical_accuracy: 0.9846 - output_8_sparse_categorical_
→accuracy: 0.9971 - output_9_sparse_categorical_accuracy: 0.9689

```

Epoch 9/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0629 - output_1_loss: 0.
→0103 - output_2_loss: 0.0687 - output_3_loss: 0.0577 - output_4_loss: 0.0321 - output_
→5_loss: 0.0671 - output_6_loss: 0.0434 - output_7_loss: 0.0470 - output_8_loss: 0.0112_
→- output_9_loss: 0.0938 - output_2_sparse_categorical_accuracy: 0.9826 - output_3_
→sparse_categorical_accuracy: 0.9869 - output_4_sparse_categorical_accuracy: 0.9921 -_
→output_5_sparse_categorical_accuracy: 0.9841 - output_6_sparse_categorical_accuracy: 0.
→9899 - output_7_sparse_categorical_accuracy: 0.9868 - output_8_sparse_categorical_
→accuracy: 0.9976 - output_9_sparse_categorical_accuracy: 0.9738

```

Epoch 10/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0562 - output_1_loss: 0.
→0092 - output_2_loss: 0.0626 - output_3_loss: 0.0507 - output_4_loss: 0.0292 - output_
→5_loss: 0.0592 - output_6_loss: 0.0383 - output_7_loss: 0.0414 - output_8_loss: 0.0099_
→- output_9_loss: 0.0842 - output_2_sparse_categorical_accuracy: 0.9835 - output_3_
→sparse_categorical_accuracy: 0.9879 - output_4_sparse_categorical_accuracy: 0.9925 -_
→output_5_sparse_categorical_accuracy: 0.9861 - output_6_sparse_categorical_accuracy: 0.
→9906 - output_7_sparse_categorical_accuracy: 0.9890 - output_8_sparse_categorical_

```

(continues on next page)

(continued from previous page)

```

→accuracy: 0.9978 - output_9_sparse_categorical_accuracy: 0.9762
Epoch 11/50
203/203 [=====] - 1s 6ms/step - loss: 0.0511 - output_1_loss: 0.
→0086 - output_2_loss: 0.0569 - output_3_loss: 0.0454 - output_4_loss: 0.0266 - output_
→5_loss: 0.0535 - output_6_loss: 0.0346 - output_7_loss: 0.0372 - output_8_loss: 0.0091_
→- output_9_loss: 0.0760 - output_2_sparse_categorical_accuracy: 0.9857 - output_3_
→sparse_categorical_accuracy: 0.9893 - output_4_sparse_categorical_accuracy: 0.9935 -_
→output_5_sparse_categorical_accuracy: 0.9883 - output_6_sparse_categorical_accuracy: 0.
→9916 - output_7_sparse_categorical_accuracy: 0.9900 - output_8_sparse_categorical_
→accuracy: 0.9980 - output_9_sparse_categorical_accuracy: 0.9783
Epoch 12/50
203/203 [=====] - 1s 6ms/step - loss: 0.0465 - output_1_loss: 0.
→0081 - output_2_loss: 0.0522 - output_3_loss: 0.0406 - output_4_loss: 0.0242 - output_
→5_loss: 0.0476 - output_6_loss: 0.0319 - output_7_loss: 0.0334 - output_8_loss: 0.0082_
→- output_9_loss: 0.0687 - output_2_sparse_categorical_accuracy: 0.9865 - output_3_
→sparse_categorical_accuracy: 0.9906 - output_4_sparse_categorical_accuracy: 0.9945 -_
→output_5_sparse_categorical_accuracy: 0.9890 - output_6_sparse_categorical_accuracy: 0.
→9925 - output_7_sparse_categorical_accuracy: 0.9913 - output_8_sparse_categorical_
→accuracy: 0.9984 - output_9_sparse_categorical_accuracy: 0.9803
Epoch 13/50
203/203 [=====] - 1s 6ms/step - loss: 0.0424 - output_1_loss: 0.
→0075 - output_2_loss: 0.0477 - output_3_loss: 0.0369 - output_4_loss: 0.0221 - output_
→5_loss: 0.0431 - output_6_loss: 0.0289 - output_7_loss: 0.0305 - output_8_loss: 0.0076_
→- output_9_loss: 0.0626 - output_2_sparse_categorical_accuracy: 0.9875 - output_3_
→sparse_categorical_accuracy: 0.9912 - output_4_sparse_categorical_accuracy: 0.9950 -_
→output_5_sparse_categorical_accuracy: 0.9903 - output_6_sparse_categorical_accuracy: 0.
→9934 - output_7_sparse_categorical_accuracy: 0.9921 - output_8_sparse_categorical_
→accuracy: 0.9987 - output_9_sparse_categorical_accuracy: 0.9820
Epoch 14/50
203/203 [=====] - 1s 6ms/step - loss: 0.0390 - output_1_loss: 0.
→0069 - output_2_loss: 0.0443 - output_3_loss: 0.0341 - output_4_loss: 0.0205 - output_
→5_loss: 0.0392 - output_6_loss: 0.0267 - output_7_loss: 0.0277 - output_8_loss: 0.0070_
→- output_9_loss: 0.0571 - output_2_sparse_categorical_accuracy: 0.9880 - output_3_
→sparse_categorical_accuracy: 0.9923 - output_4_sparse_categorical_accuracy: 0.9953 -_
→output_5_sparse_categorical_accuracy: 0.9908 - output_6_sparse_categorical_accuracy: 0.
→9936 - output_7_sparse_categorical_accuracy: 0.9930 - output_8_sparse_categorical_
→accuracy: 0.9986 - output_9_sparse_categorical_accuracy: 0.9840
Epoch 15/50
203/203 [=====] - 1s 6ms/step - loss: 0.0362 - output_1_loss: 0.
→0066 - output_2_loss: 0.0412 - output_3_loss: 0.0316 - output_4_loss: 0.0191 - output_
→5_loss: 0.0352 - output_6_loss: 0.0249 - output_7_loss: 0.0249 - output_8_loss: 0.0064_
→- output_9_loss: 0.0531 - output_2_sparse_categorical_accuracy: 0.9891 - output_3_
→sparse_categorical_accuracy: 0.9922 - output_4_sparse_categorical_accuracy: 0.9957 -_
→output_5_sparse_categorical_accuracy: 0.9919 - output_6_sparse_categorical_accuracy: 0.
→9942 - output_7_sparse_categorical_accuracy: 0.9937 - output_8_sparse_categorical_
→accuracy: 0.9987 - output_9_sparse_categorical_accuracy: 0.9852
Epoch 16/50
203/203 [=====] - 1s 6ms/step - loss: 0.0336 - output_1_loss: 0.
→0064 - output_2_loss: 0.0380 - output_3_loss: 0.0294 - output_4_loss: 0.0178 - output_
→5_loss: 0.0322 - output_6_loss: 0.0226 - output_7_loss: 0.0224 - output_8_loss: 0.0060_
→- output_9_loss: 0.0491 - output_2_sparse_categorical_accuracy: 0.9906 - output_3_

```

(continues on next page)

(continued from previous page)

```

→sparse_categorical_accuracy: 0.9932 - output_4_sparse_categorical_accuracy: 0.9963 -
→output_5_sparse_categorical_accuracy: 0.9928 - output_6_sparse_categorical_accuracy: 0.
→9946 - output_7_sparse_categorical_accuracy: 0.9943 - output_8_sparse_categorical_
→accuracy: 0.9987 - output_9_sparse_categorical_accuracy: 0.9865
Epoch 17/50
203/203 [=====] - 1s 6ms/step - loss: 0.0313 - output_1_loss: 0.
→0062 - output_2_loss: 0.0353 - output_3_loss: 0.0271 - output_4_loss: 0.0166 - output_
→5_loss: 0.0294 - output_6_loss: 0.0214 - output_7_loss: 0.0205 - output_8_loss: 0.0055
→- output_9_loss: 0.0456 - output_2_sparse_categorical_accuracy: 0.9910 - output_3_
→sparse_categorical_accuracy: 0.9936 - output_4_sparse_categorical_accuracy: 0.9965 -
→output_5_sparse_categorical_accuracy: 0.9935 - output_6_sparse_categorical_accuracy: 0.
→9949 - output_7_sparse_categorical_accuracy: 0.9947 - output_8_sparse_categorical_
→accuracy: 0.9990 - output_9_sparse_categorical_accuracy: 0.9879
Epoch 18/50
203/203 [=====] - 1s 6ms/step - loss: 0.0292 - output_1_loss: 0.
→0058 - output_2_loss: 0.0327 - output_3_loss: 0.0253 - output_4_loss: 0.0155 - output_
→5_loss: 0.0272 - output_6_loss: 0.0198 - output_7_loss: 0.0188 - output_8_loss: 0.0052
→- output_9_loss: 0.0428 - output_2_sparse_categorical_accuracy: 0.9913 - output_3_
→sparse_categorical_accuracy: 0.9939 - output_4_sparse_categorical_accuracy: 0.9968 -
→output_5_sparse_categorical_accuracy: 0.9938 - output_6_sparse_categorical_accuracy: 0.
→9955 - output_7_sparse_categorical_accuracy: 0.9955 - output_8_sparse_categorical_
→accuracy: 0.9989 - output_9_sparse_categorical_accuracy: 0.9888
Epoch 19/50
203/203 [=====] - 1s 7ms/step - loss: 0.0275 - output_1_loss: 0.
→0057 - output_2_loss: 0.0303 - output_3_loss: 0.0241 - output_4_loss: 0.0146 - output_
→5_loss: 0.0249 - output_6_loss: 0.0186 - output_7_loss: 0.0173 - output_8_loss: 0.0047
→- output_9_loss: 0.0397 - output_2_sparse_categorical_accuracy: 0.9919 - output_3_
→sparse_categorical_accuracy: 0.9941 - output_4_sparse_categorical_accuracy: 0.9969 -
→output_5_sparse_categorical_accuracy: 0.9940 - output_6_sparse_categorical_accuracy: 0.
→9955 - output_7_sparse_categorical_accuracy: 0.9960 - output_8_sparse_categorical_
→accuracy: 0.9991 - output_9_sparse_categorical_accuracy: 0.9896
Epoch 20/50
203/203 [=====] - 1s 6ms/step - loss: 0.0254 - output_1_loss: 0.
→0054 - output_2_loss: 0.0281 - output_3_loss: 0.0218 - output_4_loss: 0.0136 - output_
→5_loss: 0.0225 - output_6_loss: 0.0170 - output_7_loss: 0.0156 - output_8_loss: 0.0045
→- output_9_loss: 0.0375 - output_2_sparse_categorical_accuracy: 0.9927 - output_3_
→sparse_categorical_accuracy: 0.9949 - output_4_sparse_categorical_accuracy: 0.9972 -
→output_5_sparse_categorical_accuracy: 0.9949 - output_6_sparse_categorical_accuracy: 0.
→9962 - output_7_sparse_categorical_accuracy: 0.9965 - output_8_sparse_categorical_
→accuracy: 0.9992 - output_9_sparse_categorical_accuracy: 0.9905
Epoch 21/50
203/203 [=====] - 1s 6ms/step - loss: 0.0239 - output_1_loss: 0.
→0052 - output_2_loss: 0.0259 - output_3_loss: 0.0204 - output_4_loss: 0.0128 - output_
→5_loss: 0.0206 - output_6_loss: 0.0158 - output_7_loss: 0.0144 - output_8_loss: 0.0041
→- output_9_loss: 0.0352 - output_2_sparse_categorical_accuracy: 0.9934 - output_3_
→sparse_categorical_accuracy: 0.9952 - output_4_sparse_categorical_accuracy: 0.9974 -
→output_5_sparse_categorical_accuracy: 0.9955 - output_6_sparse_categorical_accuracy: 0.
→9962 - output_7_sparse_categorical_accuracy: 0.9967 - output_8_sparse_categorical_
→accuracy: 0.9992 - output_9_sparse_categorical_accuracy: 0.9910
Epoch 22/50
203/203 [=====] - ETA: 0s - loss: 0.0228 - output_1_loss: 0.
→0051 - output_2_loss: 0.0247 - output_3_loss: 0.0194 - output_4_loss: 0.0124 - output_

```

(continues on next page)

(continued from previous page)

```

→5_loss: 0.0195 - output_6_loss: 0.0149 - output_7_loss: 0.0135 - output_8_loss: 0.0040_
→- output_9_loss: 0.0327 - output_2_sparse_categorical_accuracy: 0.9938 - output_3_
→sparse_categorical_accuracy: 0.9951 - output_4_sparse_categorical_accuracy: 0.9975 -_
→output_5_sparse_categorical_accuracy: 0.9957 - output_6_sparse_categorical_accuracy: 0.
→9966 - output_7_sparse_categorical_accuracy: 0.9970 - output_8_sparse_categorical_
→accuracy: 0.9990 - output_9_sparse_categorical_accuracy: 0.991 - 1s 6ms/step - loss: 0.
→0227 - output_1_loss: 0.0051 - output_2_loss: 0.0244 - output_3_loss: 0.0193 - output_
→4_loss: 0.0124 - output_5_loss: 0.0192 - output_6_loss: 0.0149 - output_7_loss: 0.0133_
→- output_8_loss: 0.0040 - output_9_loss: 0.0332 - output_2_sparse_categorical_accuracy:
→ 0.9939 - output_3_sparse_categorical_accuracy: 0.9952 - output_4_sparse_categorical_
→accuracy: 0.9975 - output_5_sparse_categorical_accuracy: 0.9957 - output_6_sparse_
→categorical_accuracy: 0.9966 - output_7_sparse_categorical_accuracy: 0.9971 - output_8_
→sparse_categorical_accuracy: 0.9991 - output_9_sparse_categorical_accuracy: 0.9914

```

Epoch 23/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0212 - output_1_loss: 0.
→0049 - output_2_loss: 0.0228 - output_3_loss: 0.0179 - output_4_loss: 0.0116 - output_
→5_loss: 0.0174 - output_6_loss: 0.0141 - output_7_loss: 0.0123 - output_8_loss: 0.0037_
→- output_9_loss: 0.0311 - output_2_sparse_categorical_accuracy: 0.9940 - output_3_
→sparse_categorical_accuracy: 0.9958 - output_4_sparse_categorical_accuracy: 0.9976 -_
→output_5_sparse_categorical_accuracy: 0.9964 - output_6_sparse_categorical_accuracy: 0.
→9969 - output_7_sparse_categorical_accuracy: 0.9974 - output_8_sparse_categorical_
→accuracy: 0.9993 - output_9_sparse_categorical_accuracy: 0.9920

```

Epoch 24/50

```

203/203 [=====] - 1s 7ms/step - loss: 0.0198 - output_1_loss: 0.
→0046 - output_2_loss: 0.0209 - output_3_loss: 0.0169 - output_4_loss: 0.0109 - output_
→5_loss: 0.0163 - output_6_loss: 0.0129 - output_7_loss: 0.0116 - output_8_loss: 0.0036_
→- output_9_loss: 0.0292 - output_2_sparse_categorical_accuracy: 0.9948 - output_3_
→sparse_categorical_accuracy: 0.9958 - output_4_sparse_categorical_accuracy: 0.9980 -_
→output_5_sparse_categorical_accuracy: 0.9968 - output_6_sparse_categorical_accuracy: 0.
→9974 - output_7_sparse_categorical_accuracy: 0.9975 - output_8_sparse_categorical_
→accuracy: 0.9992 - output_9_sparse_categorical_accuracy: 0.9928

```

Epoch 25/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0190 - output_1_loss: 0.
→0046 - output_2_loss: 0.0197 - output_3_loss: 0.0157 - output_4_loss: 0.0107 - output_
→5_loss: 0.0154 - output_6_loss: 0.0123 - output_7_loss: 0.0106 - output_8_loss: 0.0031_
→- output_9_loss: 0.0278 - output_2_sparse_categorical_accuracy: 0.9950 - output_3_
→sparse_categorical_accuracy: 0.9960 - output_4_sparse_categorical_accuracy: 0.9979 -_
→output_5_sparse_categorical_accuracy: 0.9968 - output_6_sparse_categorical_accuracy: 0.
→9973 - output_7_sparse_categorical_accuracy: 0.9978 - output_8_sparse_categorical_
→accuracy: 0.9994 - output_9_sparse_categorical_accuracy: 0.9930

```

Epoch 26/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0178 - output_1_loss: 0.
→0043 - output_2_loss: 0.0186 - output_3_loss: 0.0148 - output_4_loss: 0.0099 - output_
→5_loss: 0.0139 - output_6_loss: 0.0114 - output_7_loss: 0.0100 - output_8_loss: 0.0032_
→- output_9_loss: 0.0260 - output_2_sparse_categorical_accuracy: 0.9955 - output_3_
→sparse_categorical_accuracy: 0.9963 - output_4_sparse_categorical_accuracy: 0.9982 -_
→output_5_sparse_categorical_accuracy: 0.9971 - output_6_sparse_categorical_accuracy: 0.
→9977 - output_7_sparse_categorical_accuracy: 0.9978 - output_8_sparse_categorical_
→accuracy: 0.9995 - output_9_sparse_categorical_accuracy: 0.9936

```

Epoch 27/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0167 - output_1_loss: 0.

```

(continues on next page)

(continued from previous page)

```

→0041 - output_2_loss: 0.0171 - output_3_loss: 0.0137 - output_4_loss: 0.0092 - output_
→5_loss: 0.0131 - output_6_loss: 0.0109 - output_7_loss: 0.0096 - output_8_loss: 0.0028_
→- output_9_loss: 0.0245 - output_2_sparse_categorical_accuracy: 0.9960 - output_3_
→sparse_categorical_accuracy: 0.9968 - output_4_sparse_categorical_accuracy: 0.9984 -_
→output_5_sparse_categorical_accuracy: 0.9974 - output_6_sparse_categorical_accuracy: 0.
→9980 - output_7_sparse_categorical_accuracy: 0.9980 - output_8_sparse_categorical_
→accuracy: 0.9995 - output_9_sparse_categorical_accuracy: 0.9938
Epoch 28/50
203/203 [=====] - 1s 6ms/step - loss: 0.0160 - output_1_loss: 0.
→0041 - output_2_loss: 0.0164 - output_3_loss: 0.0130 - output_4_loss: 0.0091 - output_
→5_loss: 0.0124 - output_6_loss: 0.0098 - output_7_loss: 0.0090 - output_8_loss: 0.0028_
→- output_9_loss: 0.0230 - output_2_sparse_categorical_accuracy: 0.9964 - output_3_
→sparse_categorical_accuracy: 0.9972 - output_4_sparse_categorical_accuracy: 0.9983 -_
→output_5_sparse_categorical_accuracy: 0.9980 - output_6_sparse_categorical_accuracy: 0.
→9982 - output_7_sparse_categorical_accuracy: 0.9981 - output_8_sparse_categorical_
→accuracy: 0.9994 - output_9_sparse_categorical_accuracy: 0.9944
Epoch 29/50
203/203 [=====] - 1s 6ms/step - loss: 0.0150 - output_1_loss: 0.
→0039 - output_2_loss: 0.0153 - output_3_loss: 0.0120 - output_4_loss: 0.0083 - output_
→5_loss: 0.0114 - output_6_loss: 0.0093 - output_7_loss: 0.0085 - output_8_loss: 0.0023_
→- output_9_loss: 0.0217 - output_2_sparse_categorical_accuracy: 0.9966 - output_3_
→sparse_categorical_accuracy: 0.9974 - output_4_sparse_categorical_accuracy: 0.9982 -_
→output_5_sparse_categorical_accuracy: 0.9980 - output_6_sparse_categorical_accuracy: 0.
→9981 - output_7_sparse_categorical_accuracy: 0.9981 - output_8_sparse_categorical_
→accuracy: 0.9996 - output_9_sparse_categorical_accuracy: 0.9946
Epoch 30/50
203/203 [=====] - 1s 6ms/step - loss: 0.0142 - output_1_loss: 0.
→0038 - output_2_loss: 0.0144 - output_3_loss: 0.0111 - output_4_loss: 0.0080 - output_
→5_loss: 0.0107 - output_6_loss: 0.0087 - output_7_loss: 0.0079 - output_8_loss: 0.0023_
→- output_9_loss: 0.0206 - output_2_sparse_categorical_accuracy: 0.9967 - output_3_
→sparse_categorical_accuracy: 0.9975 - output_4_sparse_categorical_accuracy: 0.9986 -_
→output_5_sparse_categorical_accuracy: 0.9981 - output_6_sparse_categorical_accuracy: 0.
→9984 - output_7_sparse_categorical_accuracy: 0.9984 - output_8_sparse_categorical_
→accuracy: 0.9997 - output_9_sparse_categorical_accuracy: 0.9949
Epoch 31/50
203/203 [=====] - 1s 7ms/step - loss: 0.0139 - output_1_loss: 0.
→0039 - output_2_loss: 0.0135 - output_3_loss: 0.0109 - output_4_loss: 0.0079 - output_
→5_loss: 0.0103 - output_6_loss: 0.0084 - output_7_loss: 0.0076 - output_8_loss: 0.0022_
→- output_9_loss: 0.0194 - output_2_sparse_categorical_accuracy: 0.9972 - output_3_
→sparse_categorical_accuracy: 0.9979 - output_4_sparse_categorical_accuracy: 0.9985 -_
→output_5_sparse_categorical_accuracy: 0.9980 - output_6_sparse_categorical_accuracy: 0.
→9984 - output_7_sparse_categorical_accuracy: 0.9984 - output_8_sparse_categorical_
→accuracy: 0.9996 - output_9_sparse_categorical_accuracy: 0.9957
Epoch 32/50
203/203 [=====] - 1s 7ms/step - loss: 0.0135 - output_1_loss: 0.
→0037 - output_2_loss: 0.0132 - output_3_loss: 0.0105 - output_4_loss: 0.0079 - output_
→5_loss: 0.0100 - output_6_loss: 0.0089 - output_7_loss: 0.0073 - output_8_loss: 0.0022_
→- output_9_loss: 0.0186 - output_2_sparse_categorical_accuracy: 0.9972 - output_3_
→sparse_categorical_accuracy: 0.9977 - output_4_sparse_categorical_accuracy: 0.9981 -_
→output_5_sparse_categorical_accuracy: 0.9986 - output_6_sparse_categorical_accuracy: 0.
→9982 - output_7_sparse_categorical_accuracy: 0.9986 - output_8_sparse_categorical_
→accuracy: 0.9995 - output_9_sparse_categorical_accuracy: 0.9958

```

(continues on next page)

(continued from previous page)

Epoch 33/50

```
203/203 [=====] - 1s 7ms/step - loss: 0.0124 - output_1_loss: 0.
→0035 - output_2_loss: 0.0120 - output_3_loss: 0.0094 - output_4_loss: 0.0069 - output_
→5_loss: 0.0089 - output_6_loss: 0.0071 - output_7_loss: 0.0071 - output_8_loss: 0.0019_
→- output_9_loss: 0.0176 - output_2_sparse_categorical_accuracy: 0.9974 - output_3_
→sparse_categorical_accuracy: 0.9980 - output_4_sparse_categorical_accuracy: 0.9986 -_
→output_5_sparse_categorical_accuracy: 0.9987 - output_6_sparse_categorical_accuracy: 0.
→9987 - output_7_sparse_categorical_accuracy: 0.9988 - output_8_sparse_categorical_
→accuracy: 0.9997 - output_9_sparse_categorical_accuracy: 0.9959
```

Epoch 34/50

```
203/203 [=====] - 1s 7ms/step - loss: 0.0128 - output_1_loss: 0.
→0037 - output_2_loss: 0.0117 - output_3_loss: 0.0095 - output_4_loss: 0.0078 - output_
→5_loss: 0.0091 - output_6_loss: 0.0094 - output_7_loss: 0.0068 - output_8_loss: 0.0022_
→- output_9_loss: 0.0166 - output_2_sparse_categorical_accuracy: 0.9976 - output_3_
→sparse_categorical_accuracy: 0.9981 - output_4_sparse_categorical_accuracy: 0.9985 -_
→output_5_sparse_categorical_accuracy: 0.9985 - output_6_sparse_categorical_accuracy: 0.
→9985 - output_7_sparse_categorical_accuracy: 0.9987 - output_8_sparse_categorical_
→accuracy: 0.9995 - output_9_sparse_categorical_accuracy: 0.9962
```

Epoch 35/50

```
203/203 [=====] - 1s 6ms/step - loss: 0.0111 - output_1_loss: 0.
→0032 - output_2_loss: 0.0108 - output_3_loss: 0.0082 - output_4_loss: 0.0062 - output_
→5_loss: 0.0079 - output_6_loss: 0.0064 - output_7_loss: 0.0063 - output_8_loss: 0.0017_
→- output_9_loss: 0.0156 - output_2_sparse_categorical_accuracy: 0.9977 - output_3_
→sparse_categorical_accuracy: 0.9983 - output_4_sparse_categorical_accuracy: 0.9988 -_
→output_5_sparse_categorical_accuracy: 0.9990 - output_6_sparse_categorical_accuracy: 0.
→9991 - output_7_sparse_categorical_accuracy: 0.9987 - output_8_sparse_categorical_
→accuracy: 0.9999 - output_9_sparse_categorical_accuracy: 0.9966
```

Epoch 36/50

```
203/203 [=====] - 1s 6ms/step - loss: 0.0108 - output_1_loss: 0.
→0033 - output_2_loss: 0.0103 - output_3_loss: 0.0078 - output_4_loss: 0.0059 - output_
→5_loss: 0.0077 - output_6_loss: 0.0061 - output_7_loss: 0.0059 - output_8_loss: 0.0016_
→- output_9_loss: 0.0147 - output_2_sparse_categorical_accuracy: 0.9981 - output_3_
→sparse_categorical_accuracy: 0.9987 - output_4_sparse_categorical_accuracy: 0.9991 -_
→output_5_sparse_categorical_accuracy: 0.9989 - output_6_sparse_categorical_accuracy: 0.
→9992 - output_7_sparse_categorical_accuracy: 0.9989 - output_8_sparse_categorical_
→accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9970
```

Epoch 37/50

```
203/203 [=====] - 1s 6ms/step - loss: 0.0102 - output_1_loss: 0.
→0031 - output_2_loss: 0.0096 - output_3_loss: 0.0075 - output_4_loss: 0.0058 - output_
→5_loss: 0.0072 - output_6_loss: 0.0058 - output_7_loss: 0.0056 - output_8_loss: 0.0016_
→- output_9_loss: 0.0139 - output_2_sparse_categorical_accuracy: 0.9982 - output_3_
→sparse_categorical_accuracy: 0.9987 - output_4_sparse_categorical_accuracy: 0.9988 -_
→output_5_sparse_categorical_accuracy: 0.9990 - output_6_sparse_categorical_accuracy: 0.
→9990 - output_7_sparse_categorical_accuracy: 0.9991 - output_8_sparse_categorical_
→accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9973
```

Epoch 38/50

```
203/203 [=====] - 1s 6ms/step - loss: 0.0098 - output_1_loss: 0.
→0030 - output_2_loss: 0.0091 - output_3_loss: 0.0071 - output_4_loss: 0.0055 - output_
→5_loss: 0.0070 - output_6_loss: 0.0053 - output_7_loss: 0.0054 - output_8_loss: 0.0015_
→- output_9_loss: 0.0131 - output_2_sparse_categorical_accuracy: 0.9985 - output_3_
→sparse_categorical_accuracy: 0.9988 - output_4_sparse_categorical_accuracy: 0.9990 -_
→
```

(continues on next page)

(continued from previous page)

```

→output_5_sparse_categorical_accuracy: 0.9990 - output_6_sparse_categorical_accuracy: 0.
→9992 - output_7_sparse_categorical_accuracy: 0.9991 - output_8_sparse_categorical_
→accuracy: 0.9999 - output_9_sparse_categorical_accuracy: 0.9972
Epoch 39/50
203/203 [=====] - 1s 6ms/step - loss: 0.0095 - output_1_loss: 0.
→0031 - output_2_loss: 0.0088 - output_3_loss: 0.0066 - output_4_loss: 0.0050 - output_
→5_loss: 0.0067 - output_6_loss: 0.0051 - output_7_loss: 0.0054 - output_8_loss: 0.0014_
→- output_9_loss: 0.0126 - output_2_sparse_categorical_accuracy: 0.9984 - output_3_
→sparse_categorical_accuracy: 0.9990 - output_4_sparse_categorical_accuracy: 0.9993 -_
→output_5_sparse_categorical_accuracy: 0.9990 - output_6_sparse_categorical_accuracy: 0.
→9993 - output_7_sparse_categorical_accuracy: 0.9990 - output_8_sparse_categorical_
→accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9975
Epoch 40/50
203/203 [=====] - 1s 6ms/step - loss: 0.0091 - output_1_loss: 0.
→0029 - output_2_loss: 0.0083 - output_3_loss: 0.0066 - output_4_loss: 0.0050 - output_
→5_loss: 0.0062 - output_6_loss: 0.0049 - output_7_loss: 0.0051 - output_8_loss: 0.0014_
→- output_9_loss: 0.0121 - output_2_sparse_categorical_accuracy: 0.9985 - output_3_
→sparse_categorical_accuracy: 0.9990 - output_4_sparse_categorical_accuracy: 0.9992 -_
→output_5_sparse_categorical_accuracy: 0.9992 - output_6_sparse_categorical_accuracy: 0.
→9993 - output_7_sparse_categorical_accuracy: 0.9990 - output_8_sparse_categorical_
→accuracy: 0.9997 - output_9_sparse_categorical_accuracy: 0.9975
Epoch 41/50
203/203 [=====] - 1s 7ms/step - loss: 0.0089 - output_1_loss: 0.
→0029 - output_2_loss: 0.0079 - output_3_loss: 0.0064 - output_4_loss: 0.0049 - output_
→5_loss: 0.0063 - output_6_loss: 0.0047 - output_7_loss: 0.0049 - output_8_loss: 0.0013_
→- output_9_loss: 0.0115 - output_2_sparse_categorical_accuracy: 0.9986 - output_3_
→sparse_categorical_accuracy: 0.9990 - output_4_sparse_categorical_accuracy: 0.9992 -_
→output_5_sparse_categorical_accuracy: 0.9991 - output_6_sparse_categorical_accuracy: 0.
→9994 - output_7_sparse_categorical_accuracy: 0.9991 - output_8_sparse_categorical_
→accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9978
Epoch 42/50
203/203 [=====] - 1s 6ms/step - loss: 0.0082 - output_1_loss: 0.
→0026 - output_2_loss: 0.0075 - output_3_loss: 0.0057 - output_4_loss: 0.0043 - output_
→5_loss: 0.0057 - output_6_loss: 0.0045 - output_7_loss: 0.0045 - output_8_loss: 0.0013_
→- output_9_loss: 0.0108 - output_2_sparse_categorical_accuracy: 0.9988 - output_3_
→sparse_categorical_accuracy: 0.9992 - output_4_sparse_categorical_accuracy: 0.9992 -_
→output_5_sparse_categorical_accuracy: 0.9994 - output_6_sparse_categorical_accuracy: 0.
→9994 - output_7_sparse_categorical_accuracy: 0.9992 - output_8_sparse_categorical_
→accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9981
Epoch 43/50
203/203 [=====] - 1s 6ms/step - loss: 0.0080 - output_1_loss: 0.
→0027 - output_2_loss: 0.0070 - output_3_loss: 0.0055 - output_4_loss: 0.0042 - output_
→5_loss: 0.0054 - output_6_loss: 0.0042 - output_7_loss: 0.0044 - output_8_loss: 0.0011_
→- output_9_loss: 0.0103 - output_2_sparse_categorical_accuracy: 0.9989 - output_3_
→sparse_categorical_accuracy: 0.9992 - output_4_sparse_categorical_accuracy: 0.9993 -_
→output_5_sparse_categorical_accuracy: 0.9994 - output_6_sparse_categorical_accuracy: 0.
→9997 - output_7_sparse_categorical_accuracy: 0.9992 - output_8_sparse_categorical_
→accuracy: 0.9999 - output_9_sparse_categorical_accuracy: 0.9981
Epoch 44/50
203/203 [=====] - 1s 6ms/step - loss: 0.0077 - output_1_loss: 0.
→0027 - output_2_loss: 0.0067 - output_3_loss: 0.0052 - output_4_loss: 0.0039 - output_
→5_loss: 0.0053 - output_6_loss: 0.0040 - output_7_loss: 0.0043 - output_8_loss: 0.0012_

```

(continues on next page)

(continued from previous page)

```

→- output_9_loss: 0.0098 - output_2_sparse_categorical_accuracy: 0.9990 - output_3_
→sparse_categorical_accuracy: 0.9992 - output_4_sparse_categorical_accuracy: 0.9992 -
→output_5_sparse_categorical_accuracy: 0.9994 - output_6_sparse_categorical_accuracy: 0.
→9995 - output_7_sparse_categorical_accuracy: 0.9993 - output_8_sparse_categorical_
→accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9982

```

Epoch 45/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0074 - output_1_loss: 0.
→0026 - output_2_loss: 0.0066 - output_3_loss: 0.0051 - output_4_loss: 0.0038 - output_
→5_loss: 0.0049 - output_6_loss: 0.0037 - output_7_loss: 0.0042 - output_8_loss: 0.0010
→- output_9_loss: 0.0094 - output_2_sparse_categorical_accuracy: 0.9990 - output_3_
→sparse_categorical_accuracy: 0.9992 - output_4_sparse_categorical_accuracy: 0.9994 -
→output_5_sparse_categorical_accuracy: 0.9995 - output_6_sparse_categorical_accuracy: 0.
→9997 - output_7_sparse_categorical_accuracy: 0.9992 - output_8_sparse_categorical_
→accuracy: 0.9999 - output_9_sparse_categorical_accuracy: 0.9984

```

Epoch 46/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0075 - output_1_loss: 0.
→0027 - output_2_loss: 0.0064 - output_3_loss: 0.0050 - output_4_loss: 0.0039 - output_
→5_loss: 0.0050 - output_6_loss: 0.0041 - output_7_loss: 0.0040 - output_8_loss: 0.0010
→- output_9_loss: 0.0091 - output_2_sparse_categorical_accuracy: 0.9989 - output_3_
→sparse_categorical_accuracy: 0.9993 - output_4_sparse_categorical_accuracy: 0.9993 -
→output_5_sparse_categorical_accuracy: 0.9994 - output_6_sparse_categorical_accuracy: 0.
→9994 - output_7_sparse_categorical_accuracy: 0.9992 - output_8_sparse_categorical_
→accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9984

```

Epoch 47/50

```

203/203 [=====] - 1s 6ms/step - loss: 0.0072 - output_1_loss: 0.
→0026 - output_2_loss: 0.0060 - output_3_loss: 0.0048 - output_4_loss: 0.0040 - output_
→5_loss: 0.0049 - output_6_loss: 0.0038 - output_7_loss: 0.0038 - output_8_loss: 0.0010
→- output_9_loss: 0.0088 - output_2_sparse_categorical_accuracy: 0.9991 - output_3_
→sparse_categorical_accuracy: 0.9993 - output_4_sparse_categorical_accuracy: 0.9992 -
→output_5_sparse_categorical_accuracy: 0.9994 - output_6_sparse_categorical_accuracy: 0.
→9996 - output_7_sparse_categorical_accuracy: 0.9994 - output_8_sparse_categorical_
→accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9986

```

Epoch 48/50

```

203/203 [=====] - 1s 7ms/step - loss: 0.0069 - output_1_loss: 0.
→0025 - output_2_loss: 0.0060 - output_3_loss: 0.0047 - output_4_loss: 0.0035 - output_
→5_loss: 0.0046 - output_6_loss: 0.0036 - output_7_loss: 0.0037 - output_8_loss: 0.0013
→- output_9_loss: 0.0086 - output_2_sparse_categorical_accuracy: 0.9991 - output_3_
→sparse_categorical_accuracy: 0.9993 - output_4_sparse_categorical_accuracy: 0.9993 -
→output_5_sparse_categorical_accuracy: 0.9995 - output_6_sparse_categorical_accuracy: 0.
→9994 - output_7_sparse_categorical_accuracy: 0.9994 - output_8_sparse_categorical_
→accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9985

```

Epoch 49/50

```

203/203 [=====] - 1s 7ms/step - loss: 0.0065 - output_1_loss: 0.
→0023 - output_2_loss: 0.0054 - output_3_loss: 0.0042 - output_4_loss: 0.0031 - output_
→5_loss: 0.0043 - output_6_loss: 0.0034 - output_7_loss: 0.0039 - output_8_loss: 9.
→7766e-04 - output_9_loss: 0.0082 - output_2_sparse_categorical_accuracy: 0.9992 -
→output_3_sparse_categorical_accuracy: 0.9994 - output_4_sparse_categorical_accuracy: 0.
→9996 - output_5_sparse_categorical_accuracy: 0.9995 - output_6_sparse_categorical_
→accuracy: 0.9997 - output_7_sparse_categorical_accuracy: 0.9993 - output_8_sparse_
→categorical_accuracy: 0.9998 - output_9_sparse_categorical_accuracy: 0.9985

```

Epoch 50/50

(continues on next page)

(continued from previous page)

```

203/203 [=====] - 1s 6ms/step - loss: 0.0066 - output_1_loss: 0.
↪0025 - output_2_loss: 0.0053 - output_3_loss: 0.0043 - output_4_loss: 0.0033 - output_
↪5_loss: 0.0044 - output_6_loss: 0.0032 - output_7_loss: 0.0036 - output_8_loss: 9.
↪2933e-04 - output_9_loss: 0.0075 - output_2_sparse_categorical_accuracy: 0.9993 -
↪output_3_sparse_categorical_accuracy: 0.9993 - output_4_sparse_categorical_accuracy: 0.
↪9993 - output_5_sparse_categorical_accuracy: 0.9995 - output_6_sparse_categorical_
↪accuracy: 0.9996 - output_7_sparse_categorical_accuracy: 0.9994 - output_8_sparse_
↪categorical_accuracy: 0.9999 - output_9_sparse_categorical_accuracy: 0.9986
INFO:tensorflow:Assets written to: tensorflow/ADULT_autoencoder/assets

```

Counterfactual with Reinforcement Learning

```

[10]: # Define constants
COEFF_SPARSITY = 0.5           # sparsity coefficient
COEFF_CONSISTENCY = 0.5       # consistency coefficient
TRAIN_STEPS = 10000          # number of training steps -> consider increasing the
↪number of steps
BATCH_SIZE = 100             # batch size

```

Define dataset specific attributes and constraints

A desirable property of a method for generating counterfactuals is to allow feature conditioning. Real-world datasets usually include immutable features such as Sex or Race, which should remain unchanged throughout the counterfactual search procedure. Similarly, a numerical feature such as Age should only increase for a counterfactual to be actionable.

```

[11]: # Define immutable features.
immutable_features = ['Marital Status', 'Relationship', 'Race', 'Sex']

# Define ranges. This means that the `Age` feature can not decrease.
ranges = {'Age': [0.0, 1.0]}

```

Define and fit the explainer

```

[12]: explainer = CounterfactualRLTabular(predictor=predictor,
                                         encoder=heae.encoder,
                                         decoder=heae.decoder,
                                         latent_dim=LATENT_DIM,
                                         encoder_preprocessor=heae_preprocessor,
                                         decoder_inv_preprocessor=heae_inv_preprocessor,
                                         coeff_sparsity=COEFF_SPARSITY,
                                         coeff_consistency=COEFF_CONSISTENCY,
                                         category_map=adult.category_map,
                                         feature_names=adult.feature_names,
                                         ranges=ranges,
                                         immutable_features=immutable_features,
                                         train_steps=TRAIN_STEPS,
                                         batch_size=BATCH_SIZE,
                                         backend="tensorflow")

```

```
[13]: # Fit the explainer.
explainer = explainer.fit(X=X_train)

100%|=====| 10000/10000 [06:37<00:00, 25.17it/s]
```

Test explainer

```
[14]: # Select some positive examples.
X_positive = X_test[np.argmax(predictor(X_test), axis=1) == 1]

X = X_positive[:1000]
Y_t = np.array([0])
C = [{"Age": [0, 20], "Workclass": ["State-gov", "?", "Local-gov"]}]
```

```
[15]: # Generate counterfactual instances.
explanation = explainer.explain(X, Y_t, C)

100%|=====| 10/10 [00:00<00:00, 34.63it/s]
```

```
[16]: # Concat labels to the original instances.
orig = np.concatenate(
    [explanation.data['orig']['X'], explanation.data['orig']['class']],
    axis=1
)

# Concat labels to the counterfactual instances.
cf = np.concatenate(
    [explanation.data['cf']['X'], explanation.data['cf']['class']],
    axis=1
)

# Define new feature names and category map by including the label.
feature_names = adult.feature_names + ["Label"]
category_map = deepcopy(adult.category_map)
category_map.update({feature_names.index("Label"): adult.target_names})

# Replace label encodings with strings.
orig_pd = pd.DataFrame(
    apply_category_mapping(orig, category_map),
    columns=feature_names
)

cf_pd = pd.DataFrame(
    apply_category_mapping(cf, category_map),
    columns=feature_names
)
```

```
[17]: orig_pd.head(n=10)
```

```
[17]:   Age   Workclass   Education Marital Status   Occupation \
0   60   Private High School grad   Married   Blue-Collar
```

(continues on next page)

(continued from previous page)

```

1 35      Private  High School grad      Married  White-Collar
2 39      State-gov      Masters      Married  Professional
3 44  Self-emp-inc  High School grad      Married      Sales
4 39      Private      Bachelors      Separated  White-Collar
5 45      Private  High School grad      Married  Blue-Collar
6 50      Private      Bachelors      Married  Professional
7 29      Private      Bachelors      Married  White-Collar
8 47      Private      Bachelors      Married  Professional
9 35      Private      Bachelors      Married  White-Collar

      Relationship  Race      Sex Capital Gain Capital Loss Hours per week \
0      Husband  White      Male      7298      0      40
1      Husband  White      Male      7688      0      50
2      Wife  White      Female      5178      0      38
3      Husband  White      Male      0      0      50
4  Not-in-family  White      Female      13550      0      50
5      Husband  White      Male      0      1902      40
6      Husband  White      Male      0      0      50
7      Wife  White      Female      0      0      50
8      Husband  White      Male      0      0      50
9      Husband  White      Male      0      0      70

      Country Label
0  United-States >50K
1  United-States >50K
2  United-States >50K
3  United-States >50K
4  United-States >50K
5      ? >50K
6  United-States >50K
7  United-States >50K
8  United-States >50K
9  United-States >50K

```

```
[18]: cf_pd.head(n=10)
```

```

[18]:  Age      Workclass      Education Marital Status      Occupation \
0  60      Private  High School grad      Married  Blue-Collar
1  35      Private      Dropout      Married  Blue-Collar
2  39      State-gov      Dropout      Married  Service
3  44  Self-emp-inc  High School grad      Married  Sales
4  39      Private      Bachelors      Separated  White-Collar
5  45      Private  High School grad      Married  Blue-Collar
6  50      Private      Dropout      Married  Service
7  29      Private      Dropout      Married  Sales
8  47      Private      Dropout      Married  Service
9  35      Private      Dropout      Married  Sales

      Relationship  Race      Sex Capital Gain Capital Loss Hours per week \
0      Husband  White      Male      320      0      40
1      Husband  White      Male      125      0      50
2      Wife  White      Female      538      15      39

```

(continues on next page)

(continued from previous page)

3	Husband	White	Male	0	0	50
4	Not-in-family	White	Female	1922	0	51
5	Husband	White	Male	0	1900	41
6	Husband	White	Male	0	0	51
7	Wife	White	Female	0	0	50
8	Husband	White	Male	0	0	51
9	Husband	White	Male	0	0	71

	Country	Label
0	United-States	<=50K
1	United-States	<=50K
2	United-States	<=50K
3	United-States	>50K
4	United-States	<=50K
5	Latin-America	>50K
6	United-States	<=50K
7	United-States	<=50K
8	United-States	<=50K
9	United-States	<=50K

Diversity

```
[19]: # Generate counterfactual instances.
X = X_positive[0].reshape(1, -1)
explanation = explainer.explain(X=X, Y_t=Y_t, C=C, diversity=True, num_samples=100,
    ↪ batch_size=10)

12it [00:00, 26.20it/s]
```

```
[20]: # Concat label column.
orig = np.concatenate(
    [explanation.data['orig']['X'], explanation.data['orig']['class']],
    axis=1
)

cf = np.concatenate(
    [explanation.data['cf']['X'], explanation.data['cf']['class']],
    axis=1
)

# Transform label encodings to string.
orig_pd = pd.DataFrame(
    apply_category_mapping(orig, category_map),
    columns=feature_names,
)

cf_pd = pd.DataFrame(
    apply_category_mapping(cf, category_map),
    columns=feature_names,
)
```

```
[21]: orig_pd.head(n=5)
```

```
[21]:   Age Workclass      Education Marital Status  Occupation Relationship \
0   60   Private  High School grad      Married  Blue-Collar      Husband

   Race  Sex Capital Gain Capital Loss Hours per week      Country Label
0  White  Male      7298           0         40  United-States  >50K
```

```
[22]: cf_pd.head(n=5)
```

```
[22]:   Age Workclass      Education Marital Status  Occupation Relationship \
0   60   Private      Dropout      Married  Blue-Collar      Husband
1   60   Private  High School grad      Married  Blue-Collar      Husband
2   60   Private  High School grad      Married  Blue-Collar      Husband
3   60   Private  High School grad      Married  Blue-Collar      Husband
4   60   Private  High School grad      Married  Blue-Collar      Husband

   Race  Sex Capital Gain Capital Loss Hours per week      Country Label
0  White  Male      143           0         40  United-States  <=50K
1  White  Male       49           0         40  United-States  <=50K
2  White  Male       84           0         40  United-States  <=50K
3  White  Male       87           0         41  United-States  <=50K
4  White  Male       97           0         40  United-States  <=50K
```

Logging

Logging is clearly important when dealing with deep learning models. Thus, we provide an interface to write custom callbacks for logging purposes after each training step which we defined [here](#). In the following cells we provide some example to log in **Weights and Biases**.

Logging reward callback

```
[23]: class RewardCallback(Callback):
        def __call__(self,
                      step: int,
                      update: int,
                      model: CounterfactualRL,
                      sample: Dict[str, np.ndarray],
                      losses: Dict[str, float]):

            if (step + update) % 100 != 0:
                return

            # get the counterfactual and target
            Y_t = sample["Y_t"]
            X_cf = model.params["decoder_inv_preprocessor"](sample["X_cf"])

            # get prediction label
            Y_m_cf = predictor(X_cf)

            # compute reward
```

(continues on next page)

(continued from previous page)

```
reward = np.mean(model.params["reward_func"](Y_m_cf, Y_t))
wandb.log({"reward": reward})
```

Logging losses callback

```
[24]: class LossCallback(Callback):
    def __call__(self,
                 step: int,
                 update: int,
                 model: CounterfactualRL,
                 sample: Dict[str, np.ndarray],
                 losses: Dict[str, float]):
        # Log training losses.
        if (step + update) % 100 == 0:
            wandb.log(losses)
```

Logging tables callback

```
[25]: class TablesCallback(Callback):
    def __call__(self,
                 step: int,
                 update: int,
                 model: CounterfactualRL,
                 sample: Dict[str, np.ndarray],
                 losses: Dict[str, float]):
        # Log every 1000 steps
        if step % 1000 != 0:
            return

        # Define number of samples to be displayed.
        NUM_SAMPLES = 5

        X = heae_inv_preprocessor(sample["X"][:NUM_SAMPLES])           # input instance
        X_cf = heae_inv_preprocessor(sample["X_cf"][:NUM_SAMPLES])     # counterfactual

        Y_m = np.argmax(sample["Y_m"][:NUM_SAMPLES], axis=1).astype(int).reshape(-1, 1)
        ↪ # input labels
        Y_t = np.argmax(sample["Y_t"][:NUM_SAMPLES], axis=1).astype(int).reshape(-1, 1)
        ↪ # target labels
        Y_m_cf = np.argmax(predictor(X_cf), axis=1).astype(int).reshape(-1, 1)
        ↪ # counterfactual labels

        # Define feature names and category map for input.
        feature_names = adult.feature_names + ["Label"]
        category_map = deepcopy(adult.category_map)
        category_map.update({feature_names.index("Label"): adult.target_names})

        # Construct input array.
```

(continues on next page)

(continued from previous page)

```

inputs = np.concatenate([X, Y_m], axis=1)
inputs = pd.DataFrame(apply_category_mapping(inputs, category_map),
                      columns=feature_names)

# Define feature names and category map for counterfactual output.
feature_names += ["Target"]
category_map.update({feature_names.index("Target"): adult.target_names})

# Construct output array.
outputs = np.concatenate([X_cf, Y_m_cf, Y_t], axis=1)
outputs = pd.DataFrame(apply_category_mapping(outputs, category_map),
                      columns=feature_names)

# Log table.
wandb.log({
    "Input": wandb.Table(dataframe=inputs),
    "Output": wandb.Table(dataframe=outputs)
})

```

Having defined the callbacks, we can define a new explainer that will include logging.

```

import wandb

# Initialize wandb.
wandb_project = "Adult Census Counterfactual with Reinforcement Learning"
wandb.init(project=wandb_project)

# Define explainer as before and include callbacks.
explainer = CounterfactualRLTabular(...,
                                   callbacks=[LossCallback(), RewardCallback(),
↳ TablesCallback()])

# Fit the explainers.
explainer = explainer.fit(X=X_train)

# Close wandb.
wandb.finish()

```

8.7.2 Counterfactual with Reinforcement Learning (CFRL) on MNIST

This method is described in [Model-agnostic and Scalable Counterfactual Explanations via Reinforcement Learning](#) and can generate counterfactual instances for any black-box model. The usual optimization procedure is transformed into a learnable process allowing to generate batches of counterfactual instances in a single forward pass even for high dimensional data. The training pipeline is model-agnostic and relies only on prediction feedback by querying the black-box model. Furthermore, the method allows target and feature conditioning.

We exemplify the use case for the TensorFlow backend. This means that all models: the autoencoder, the actor and the critic are TensorFlow models. Our implementation supports PyTorch backend as well.

CFRL uses [Deep Deterministic Policy Gradient \(DDPG\)](#) by interleaving a state-action function approximator called critic, with a learning an approximator called actor to predict the optimal action. The method assumes that the critic is differentiable with respect to the action argument, thus allowing to optimize the actor's parameters efficiently through gradient-based methods.

The DDPG algorithm requires two separate networks, an actor μ and a critic Q . Given the encoded representation of the input instance $z = enc(x)$, the model prediction y_M , the target prediction y_T and the conditioning vector c , the actor outputs the counterfactual's latent representation $z_{CF} = \mu(z, y_M, y_T, c)$. The decoder then projects the embedding z_{CF} back to the original input space, followed by optional post-processing.

The training step consists of simultaneously optimizing the actor and critic networks. The critic regresses on the reward R determined by the model prediction, while the actor maximizes the critic's output for the given instance through L_{max} . The actor also minimizes two objectives to encourage the generation of sparse, in-distribution counterfactuals. The sparsity loss $L_{sparsity}$ operates on the decoded counterfactual x_{CF} and combines the L_1 loss over the standardized numerical features and the L_0 loss over the categorical ones. The consistency loss $L_{consist}$ aims to encode the counterfactual x_{CF} back to the same latent representation where it was decoded from and helps to produce in-distribution counterfactual instances. Formally, the actor's loss can be written as: $L_{actor} = L_{max} + \lambda_1 L_{sparsity} + \lambda_2 L_{consistency}$

Note

To enable support for CounterfactualRLTabular with tensorflow backend, you may need to run

```
pip install alibi[tensorflow]
```

```
[2]: import os
import numpy as np
import matplotlib.pyplot as plt
from typing import Dict

import tensorflow as tf
import tensorflow.keras as keras

from alibi.explainers import CounterfactualRL
from alibi.models.tensorflow import AE
from alibi.models.tensorflow import Actor, Critic
from alibi.models.tensorflow import MNISTEncoder, MNISTDecoder, MNISTClassifier
from alibi.explainers.cfml_base import Callback
```

Load MNIST dataset

```
[3]: # Define constants.
BATCH_SIZE = 64
BUFFER_SIZE = 1024

# Load MNIST dataset.
(X_train, Y_train), (X_test, Y_test) = tf.keras.datasets.mnist.load_data()

# Expand dimensions and normalize.
X_train = np.expand_dims(X_train, axis=-1).astype(np.float32) / 255.
X_test = np.expand_dims(X_test, axis=-1).astype(np.float32) / 255.

# Define trainset.
trainset_classifier = tf.data.Dataset.from_tensor_slices((X_train, Y_train))
trainset_classifier = trainset_classifier.shuffle(buffer_size=BUFFER_SIZE).batch(BATCH_
    →SIZE)
```

(continues on next page)

(continued from previous page)

```
# Define testset.
testset_classifier = tf.data.Dataset.from_tensor_slices((X_test, Y_test))
testset_classifier = testset_classifier.shuffle(buffer_size=BUFFER_SIZE).batch(BATCH_
↳ SIZE)
```

Define and train CNN classifier

```
[4]: # Number of classes.
NUM_CLASSES = 10
EPOCHS = 5

# Define classifier path and create dir if it doesn't exist.
classifier_path = os.path.join("tensorflow", "MNIST_classifier")
if not os.path.exists(classifier_path):
    os.makedirs(classifier_path)

# Construct classifier. This is the classifier used in the paper experiments.
classifier = MNISTClassifier(output_dim=NUM_CLASSES)

# Define optimizer and loss function
optimizer = keras.optimizers.Adam(learning_rate=1e-3)
loss = keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# Compile the model.
classifier.compile(optimizer=optimizer,
                  loss=loss,
                  metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])

if len(os.listdir(classifier_path)) == 0:
    # Fit and save the classifier.
    classifier.fit(trainset_classifier, epochs=EPOCHS)
    classifier.save(classifier_path)
else:
    # Load the classifier if already fitted.
    classifier = keras.models.load_model(classifier_path)

Epoch 1/5
938/938 [=====] - 12s 11ms/step - loss: 0.4742 - sparse_
↳ categorical_accuracy: 0.8550
Epoch 2/5
938/938 [=====] - 12s 13ms/step - loss: 0.0935 - sparse_
↳ categorical_accuracy: 0.9709
Epoch 3/5
938/938 [=====] - 11s 12ms/step - loss: 0.0639 - sparse_
↳ categorical_accuracy: 0.9799
Epoch 4/5
938/938 [=====] - 11s 12ms/step - loss: 0.0516 - sparse_
↳ categorical_accuracy: 0.9832
Epoch 5/5
938/938 [=====] - 11s 12ms/step - loss: 0.0453 - sparse_
```

(continues on next page)

(continued from previous page)

```
↪categorical_accuracy: 0.9851
INFO:tensorflow:Assets written to: tensorflow/MNIST_classifier/assets
```

```
[5]: # Evaluate the classifier
loss, accuracy = classifier.evaluate(testset_classifier)

157/157 [=====] - 1s 6ms/step - loss: 0.0383 - sparse_
↪categorical_accuracy: 0.9879
```

Define the predictor (black-box)

Now that we've trained the CNN classifier, we can define the black-box model. Note that the output of the black-box is a distribution which can be either a soft-label distribution (probabilities/logits for each class) or a hard-label distribution (one-hot encoding). Internally, CFRL takes the `argmax`. Moreover the output **DOES NOT HAVE TO BE DIFFERENTIABLE**.

```
[6]: # Define predictor function (black-box) used to train the CFRL
def predictor(X: np.ndarray):
    Y = classifier(X).numpy()
    return Y
```

Define and train autoencoder

Instead of directly modeling the perturbation vector in the potentially high-dimensional input space, we first train an autoencoder. The weights of the encoder are frozen and the actor applies the counterfactual perturbations in the latent space of the encoder. The pre-trained decoder maps the counterfactual embedding back to the input feature space.

The autoencoder follows a standard design. The model is composed from two submodules, the encoder and the decoder. The forward pass consists of passing the input to the encoder, obtain the input embedding and pass the embedding through the decoder.

```
class AE(keras.Model):
    def __init__(self, encoder: keras.Model, decoder: keras.Model, **kwargs) -> None:
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def call(self, x: tf.Tensor, **kwargs):
        z = self.encoder(x)
        x_hat = self.decoder(z)
        return x_hat
```

```
[7]: # Define autoencoder trainset.
trainset_ae = tf.data.Dataset.from_tensor_slices(X_train)
trainset_ae = trainset_ae.map(lambda x: (x, x))
trainset_ae = trainset_ae.shuffle(buffer_size=BUFFER_SIZE).batch(BATCH_SIZE)

# Define autoencoder testset.
testset_ae = tf.data.Dataset.from_tensor_slices(X_test)
```

(continues on next page)

(continued from previous page)

```
testset_ae = testset_ae.map(lambda x: (x, x))
testset_ae = testset_ae.shuffle(buffer_size=BUFFER_SIZE).batch(BATCH_SIZE)
```

```
[8]: # Define autoencoder path and create dir if it doesn't exist.
ae_path = os.path.join("tensorflow", "MNIST_autoencoder")
if not os.path.exists(ae_path):
    os.makedirs(ae_path)

# Define latent dimension.
LATENT_DIM = 64
EPOCHS = 50

# Define autoencoder.
ae = AE(encoder=MNISTEncoder(latent_dim=LATENT_DIM),
        decoder=MNISTDecoder())

# Define optimizer and loss function.
optimizer = keras.optimizers.Adam(learning_rate=1e-3)
loss = keras.losses.BinaryCrossentropy(from_logits=False)

# Compile autoencoder.
ae.compile(optimizer=optimizer, loss=loss)

if len(os.listdir(ae_path)) == 0:
    # Fit and save autoencoder.
    ae.fit(trainset_ae, epochs=EPOCHS)
    ae.save(ae_path)
else:
    # Load the model.
    ae = keras.models.load_model(ae_path)
```

```
Epoch 1/50
938/938 [=====] - 9s 8ms/step - loss: 0.2846
Epoch 2/50
938/938 [=====] - 8s 8ms/step - loss: 0.1431
Epoch 3/50
938/938 [=====] - 8s 8ms/step - loss: 0.1287
Epoch 4/50
938/938 [=====] - 8s 8ms/step - loss: 0.1224
Epoch 5/50
938/938 [=====] - 8s 8ms/step - loss: 0.1184
Epoch 6/50
938/938 [=====] - 8s 8ms/step - loss: 0.1155
Epoch 7/50
938/938 [=====] - 8s 8ms/step - loss: 0.1129
Epoch 8/50
938/938 [=====] - 8s 8ms/step - loss: 0.1111
Epoch 9/50
938/938 [=====] - 8s 8ms/step - loss: 0.1094
Epoch 10/50
938/938 [=====] - 8s 8ms/step - loss: 0.1078
Epoch 11/50
```

(continues on next page)

(continued from previous page)

```

938/938 [=====] - 8s 8ms/step - loss: 0.1066
Epoch 12/50
938/938 [=====] - 8s 8ms/step - loss: 0.1056
Epoch 13/50
938/938 [=====] - 8s 8ms/step - loss: 0.1045
Epoch 14/50
938/938 [=====] - 8s 8ms/step - loss: 0.1038
Epoch 15/50
938/938 [=====] - 8s 8ms/step - loss: 0.1030
Epoch 16/50
938/938 [=====] - 8s 8ms/step - loss: 0.1023
Epoch 17/50
938/938 [=====] - 8s 8ms/step - loss: 0.1016
Epoch 18/50
938/938 [=====] - 8s 8ms/step - loss: 0.1010
Epoch 19/50
938/938 [=====] - 8s 8ms/step - loss: 0.1005
Epoch 20/50
938/938 [=====] - 8s 8ms/step - loss: 0.1000
Epoch 21/50
938/938 [=====] - 8s 8ms/step - loss: 0.0997
Epoch 22/50
938/938 [=====] - 8s 8ms/step - loss: 0.0992
Epoch 23/50
938/938 [=====] - 8s 8ms/step - loss: 0.0987
Epoch 24/50
938/938 [=====] - 8s 8ms/step - loss: 0.0983
Epoch 25/50
938/938 [=====] - 7s 7ms/step - loss: 0.0980
Epoch 26/50
938/938 [=====] - 7s 8ms/step - loss: 0.0977
Epoch 27/50
938/938 [=====] - 7s 8ms/step - loss: 0.0973
Epoch 28/50
938/938 [=====] - 7s 7ms/step - loss: 0.0971
Epoch 29/50
938/938 [=====] - 7s 7ms/step - loss: 0.0967
Epoch 30/50
938/938 [=====] - 7s 7ms/step - loss: 0.0964
Epoch 31/50
938/938 [=====] - 7s 7ms/step - loss: 0.0962
Epoch 32/50
938/938 [=====] - 7s 7ms/step - loss: 0.0960
Epoch 33/50
938/938 [=====] - 7s 8ms/step - loss: 0.0957
Epoch 34/50
938/938 [=====] - 8s 9ms/step - loss: 0.0954
Epoch 35/50
938/938 [=====] - 7s 8ms/step - loss: 0.0953
Epoch 36/50
938/938 [=====] - 8s 8ms/step - loss: 0.0951
Epoch 37/50

```

(continues on next page)

(continued from previous page)

```

938/938 [=====] - 7s 8ms/step - loss: 0.0949
Epoch 38/50
938/938 [=====] - 7s 8ms/step - loss: 0.0948
Epoch 39/50
938/938 [=====] - 7s 8ms/step - loss: 0.0944
Epoch 40/50
938/938 [=====] - 9s 10ms/step - loss: 0.0941
Epoch 41/50
938/938 [=====] - 7s 7ms/step - loss: 0.0940
Epoch 42/50
938/938 [=====] - 8s 9ms/step - loss: 0.0938
Epoch 43/50
938/938 [=====] - 9s 10ms/step - loss: 0.0937
Epoch 44/50
938/938 [=====] - 7s 7ms/step - loss: 0.0935
Epoch 45/50
938/938 [=====] - 7s 8ms/step - loss: 0.0933
Epoch 46/50
938/938 [=====] - 7s 7ms/step - loss: 0.0932
Epoch 47/50
938/938 [=====] - 7s 7ms/step - loss: 0.0930
Epoch 48/50
938/938 [=====] - 7s 7ms/step - loss: 0.0929
Epoch 49/50
938/938 [=====] - 8s 8ms/step - loss: 0.0928
Epoch 50/50
938/938 [=====] - 7s 8ms/step - loss: 0.0925
INFO:tensorflow:Assets written to: tensorflow/MNIST_autoencoder/assets

```

Test the autoencoder

```

[9]: # Define number of samples to be displayed
NUM_SAMPLES = 5

# Get some random samples from test
np.random.seed(0)
indices = np.random.choice(X_test.shape[0], NUM_SAMPLES)
inputs = [X_test[i].reshape(1, 28, 28, 1) for i in indices]
inputs = np.concatenate(inputs, axis=0)

# Pass samples through the autoencoder
inputs_hat = ae(inputs).numpy()

[10]: # Plot inputs and reconstructions.
plt.rcParams.update({'font.size': 22})
fig, ax = plt.subplots(2, NUM_SAMPLES, figsize=(25, 10))

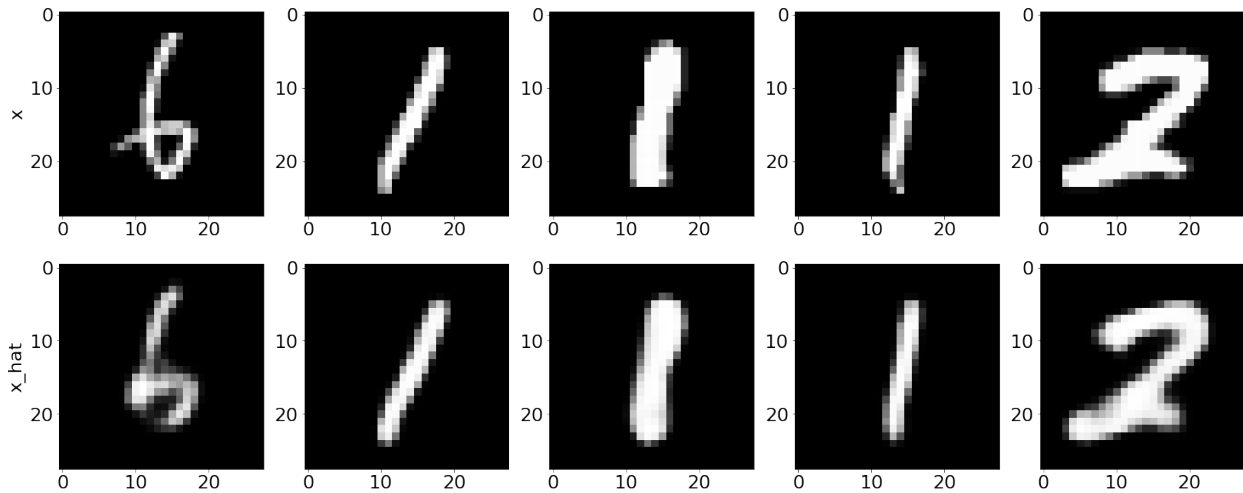
for i in range(NUM_SAMPLES):
    ax[0][i].imshow(inputs[i], cmap='gray')
    ax[1][i].imshow(inputs_hat[i], cmap='gray')

```

(continues on next page)

(continued from previous page)

```
text1 = ax[0][0].set_ylabel("x")
text2 = ax[1][0].set_ylabel("x_hat")
```



Counterfactual with Reinforcement Learning

```
[11]: # Define constants
COEFF_SPARSITY = 7.5           # sparsity coefficient
COEFF_CONSISTENCY = 0         # consistency coefficient -> no consistency
TRAIN_STEPS = 50000          # number of training steps -> consider increasing the
    ↪ number of steps
BATCH_SIZE = 100             # batch size
```

Define and fit the explainer

```
[12]: # Define explainer.
explainer = CounterfactualRL(predictor=predictor,
                             encoder=ae.encoder,
                             decoder=ae.decoder,
                             latent_dim=LATENT_DIM,
                             coeff_sparsity=COEFF_SPARSITY,
                             coeff_consistency=COEFF_CONSISTENCY,
                             train_steps=TRAIN_STEPS,
                             batch_size=BATCH_SIZE,
                             backend="tensorflow")
```

```
[13]: # Fit the explainer
explainer = explainer.fit(X=X_train)

100%|=====| 50000/50000 [31:04<00:00, 26.82it/s]
```

Test explainer

```
[14]: # Generate counterfactuals for some test instances.
explanation = explainer.explain(X_test[0:200], Y_t=np.array([2]), batch_size=100)
```

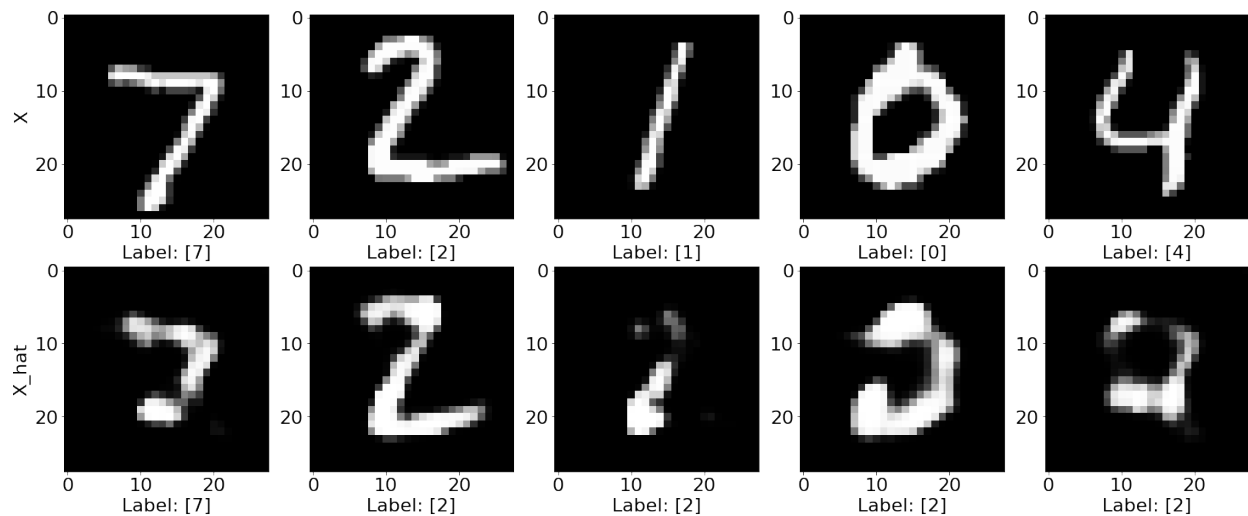
```
100%|=====| 2/2 [00:00<00:00, 36.14it/s]
```

```
[15]: fig, ax = plt.subplots(2, NUM_SAMPLES, figsize=(25, 10))

for i in range(NUM_SAMPLES):
    ax[0][i].imshow(explanation.data['orig']['X'][i], cmap='gray')
    ax[1][i].imshow(explanation.data['cf']['X'][i], cmap='gray')

    ax[0][i].set_xlabel("Label: " + str(explanation.data['orig']['class'][i]))
    ax[1][i].set_xlabel("Label: " + str(explanation.data['cf']['class'][i]))

text1 = ax[0][0].set_ylabel("X")
text2 = ax[1][0].set_ylabel("X_hat")
```



Logging

Logging is clearly important when dealing with deep learning models. Thus, we provide an interface to write custom callbacks for logging purposes after each training step which we defined [here](#). In the following cells we provide some example to log in **Weights and Biases**.

Logging reward callback

```
[16]: class RewardCallback(Callback):
    def __call__(self,
                 step: int,
                 update: int,
                 model: CounterfactualRL,
                 sample: Dict[str, np.ndarray],
                 losses: Dict[str, float]):
        if step % 100 != 0:
            return

        # Get the counterfactual and target.
        X_cf = sample["X_cf"]
        Y_t = sample["Y_t"]

        # Get prediction label.
        Y_m_cf = predictor(X_cf)

        # Compute reward
        reward = np.mean(model.params["reward_func"](Y_m_cf, Y_t))
        wandb.log({"reward": reward})
```

Logging images callback

```
[17]: class ImagesCallback(Callback):
    def __call__(self,
                 step: int,
                 update: int,
                 model: CounterfactualRL,
                 sample: Dict[str, np.ndarray],
                 losses: Dict[str, float]):
        # Log every 100 steps
        if step % 100 != 0:
            return

        # Define number of samples to be displayed.
        NUM_SAMPLES = 10

        X = sample["X"][:NUM_SAMPLES]          # input instance
        X_cf = sample["X_cf"][:NUM_SAMPLES]      # counterfactual
        diff = np.abs(X - X_cf)                  # differences

        Y_m = sample["Y_m"][:NUM_SAMPLES].astype(int) # input labels
        Y_t = sample["Y_t"][:NUM_SAMPLES].astype(int) # target labels
        Y_m_cf = predictor(X_cf).astype(int)       # counterfactual labels

        # Concatenate images,
        X = np.concatenate(X, axis=1)
        X_cf = np.concatenate(X_cf, axis=1)
        diff = np.concatenate(diff, axis=1)
```

(continues on next page)

(continued from previous page)

```

# Construct full image.
img = np.concatenate([X, X_cf, diff], axis=0)

# Construct caption.
caption = ""
caption += "Input:\t%s\n" % str(list(np.argmax(Y_m, axis=1)))
caption += "Target:\t%s\n" % str(list(np.argmax(Y_t, axis=1)))
caption += "Predicted:\t%s\n" % str(list(np.argmax(Y_m_cf, axis=1)))

# Log image.
wandb.log({"samples": wandb.Image(img, caption=caption)})

```

Logging losses callback

```

[18]: class LossCallback(Callback):
    def __call__(self,
                 step: int,
                 update: int,
                 model: CounterfactualRL,
                 sample: Dict[str, np.ndarray],
                 losses: Dict[str, float]):
        # Log every 100 updates.
        if (step + update) % 100 == 0:
            wandb.log(losses)

```

Having defined the callbacks, we can define a new explainer that will include logging.

```

import wandb

# Initialize wandb.
wandb_project = "MNIST Counterfactual with Reinforcement Learning"
wandb.init(project=wandb_project)

# Define explainer as before and include callbacks.
explainer = CounterfactualRL(...,
                             callbacks=[RewardCallback(), ImagesCallback()])

# Fit the explainer.
explainer.fit(X=X_train)

# Close wandb.
wandb.finish()

```

8.8 Integrated Gradients

8.8.1 Integrated gradients for a ResNet model trained on Imagenet dataset

In this notebook we apply the integrated gradients method to a pretrained ResNet model trained on the Imagenet data set. Integrated gradients defines an attribution value for each feature (in this case for each pixel and channel in the image) by integrating the model's gradients with respect to the input along a straight path from a baseline instance x' to the input instance x .

A more detailed description of the method can be found [here](#). Integrated gradients was originally proposed in Sundararajan et al., “Axiomatic Attribution for Deep Networks”

Note

To enable support for IntegratedGradients, you may need to run

```
pip install alibi[tensorflow]
```

```
[1]: import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from alibi.explainers import IntegratedGradients
from tensorflow.keras.applications.resnet_v2 import ResNet50V2
from alibi.datasets import load_cats
from alibi.utils import visualize_image_attr
print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # True
```

```
TF version: 2.8.0
Eager execution enabled: True
```

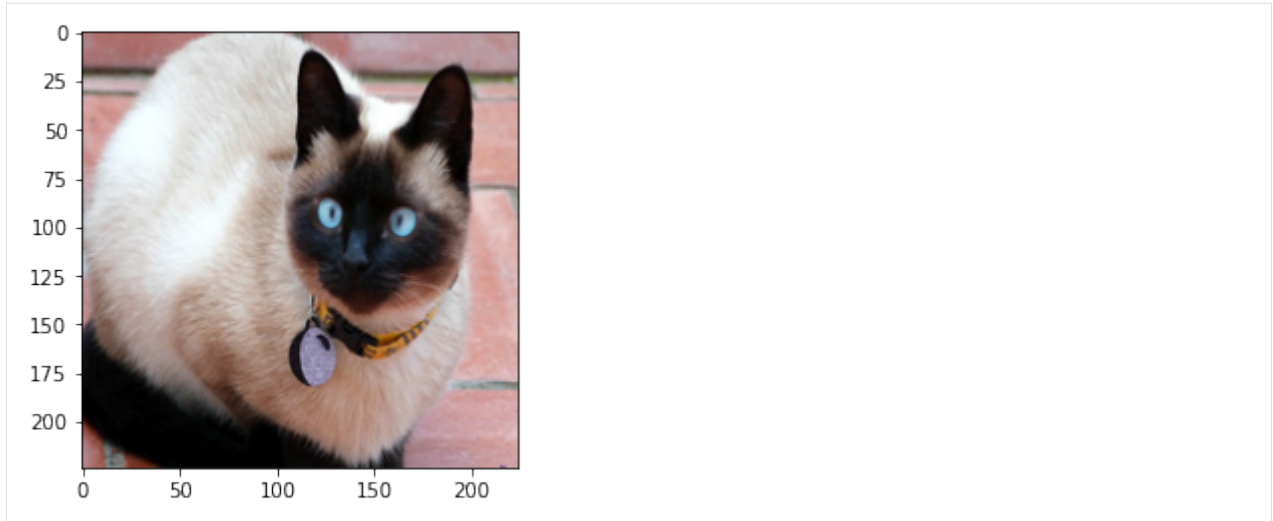
Load data

The `load_cats` function loads a small sample of images of various cat breeds.

```
[3]: image_shape = (224, 224, 3)
data, labels = load_cats(target_size=image_shape[:2], return_X_y=True)
print(f'Images shape: {data.shape}')
data = (data / 255).astype('float32')
```

```
Images shape: (4, 224, 224, 3)
```

```
[4]: i = 2
plt.imshow(data[i]);
```



Load model

Load a pretrained tensorflow model with a ResNet architecture trained on the Imagenet dataset.

```
[5]: model = ResNet50V2(weights='imagenet')
```

Calculate integrated gradients

The IntegratedGradients class implements the integrated gradients features attributions method. A description of the method can be found [here](#).

In the first example, the baselines (i.e. the starting points of the path integral) are black images (all pixel values are set to zero). This means that black areas of the image will always have zero attributions. In the second example we consider random uniform noise baselines. The path integral is defined as a straight line from the baseline to the input image. The path is approximated by choosing 50 discrete steps according to the Gauss-Legendre method.

```
[6]: n_steps = 50
method = "gausslegendre"
internal_batch_size = 50
ig = IntegratedGradients(model,
                          n_steps=n_steps,
                          method=method,
                          internal_batch_size=internal_batch_size)
```

Here we compute attributions for a single image, but batch explanations are supported (leading dimension assumed to be batch).

```
[7]: instance = np.expand_dims(data[i], axis=0)
predictions = model(instance).numpy().argmax(axis=1)
explanation = ig.explain(instance,
                       baselines=None,
                       target=predictions)
```

```
[8]: # Metadata from the explanation object
explanation.meta
```

```
[8]: {'name': 'IntegratedGradients',
      'type': ['whitebox'],
      'explanations': ['local'],
      'params': {'method': 'gausslegendre',
                  'n_steps': 50,
                  'internal_batch_size': 50,
                  'layer': 0}}
```

```
[9]: # Data fields from the explanation object
      explanation.data.keys()
```

```
[9]: dict_keys(['attributions', 'X', 'forward_kwargs', 'baselines', 'predictions', 'deltas',
               ↪ 'target'])
```

```
[10]: # Get attributions values from the explanation object
        attrs = explanation.attributions[0]
```

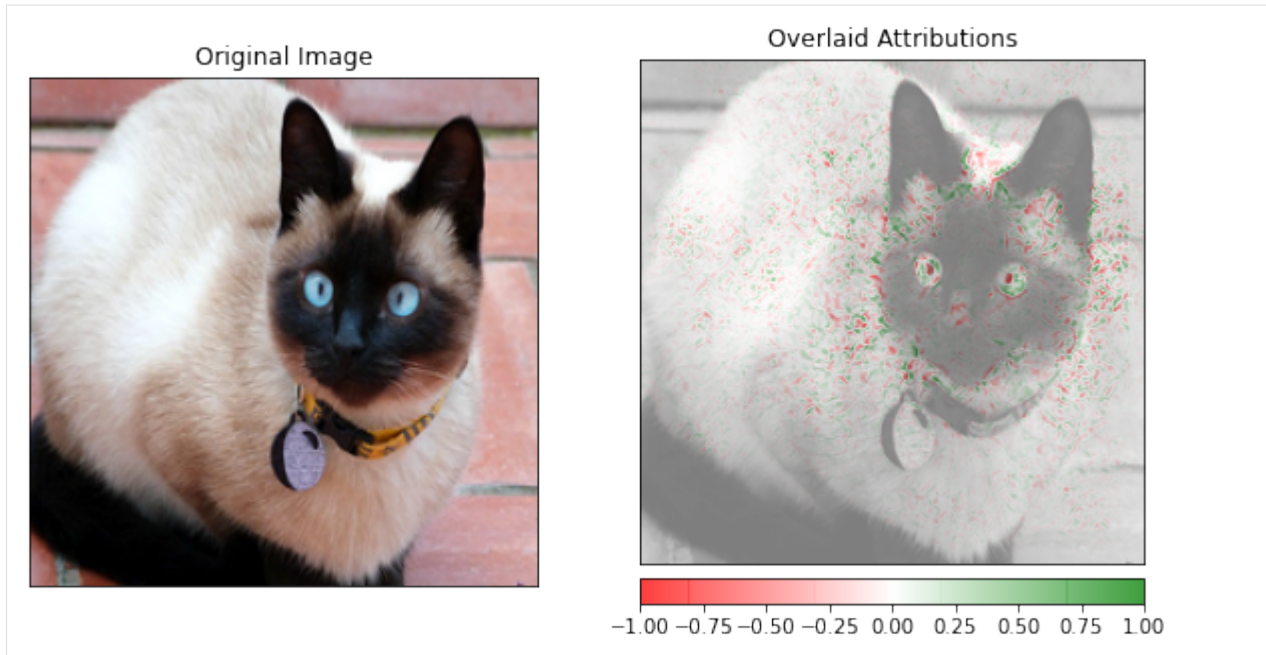
Visualize attributions

Black image baseline

Sample image from the test set and its attributions. The attributions are shown by overlaying the attributions values for each pixel to the original image. The attribution value for a pixel is obtained by summing up the attributions values for the three color channels. The attributions are scaled in a $[-1, 1]$ range: red pixels represent negative attributions, while green pixels represent positive attributions. The original image is shown in gray scale for clarity.

```
[11]: fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))
        visualize_image_attr(attr=None, original_image=data[i], method='original_image',
                              title='Original Image', plt_fig_axis=(fig, ax[0]), use_pyplot=False);

        visualize_image_attr(attr=attrs.squeeze(), original_image=data[i], method='blended_heat_
        ↪map',
                              sign='all', show_colorbar=True, title='Overlaid Attributions',
                              plt_fig_axis=(fig, ax[1]), use_pyplot=True);
```



Random baselines

Here we show the attributions obtained choosing random uniform noise as a baseline. It can be noticed that the attributions can be considerably different from the previous example, where the black image is taken as a baseline. An extensive discussion about the impact of the baselines on integrated gradients attributions can be found in P. Sturmfels at al., “[Visualizing the Impact of Feature Attribution Baselines](#)”.

```
[12]: baselines = np.random.random_sample(instance.shape)
```

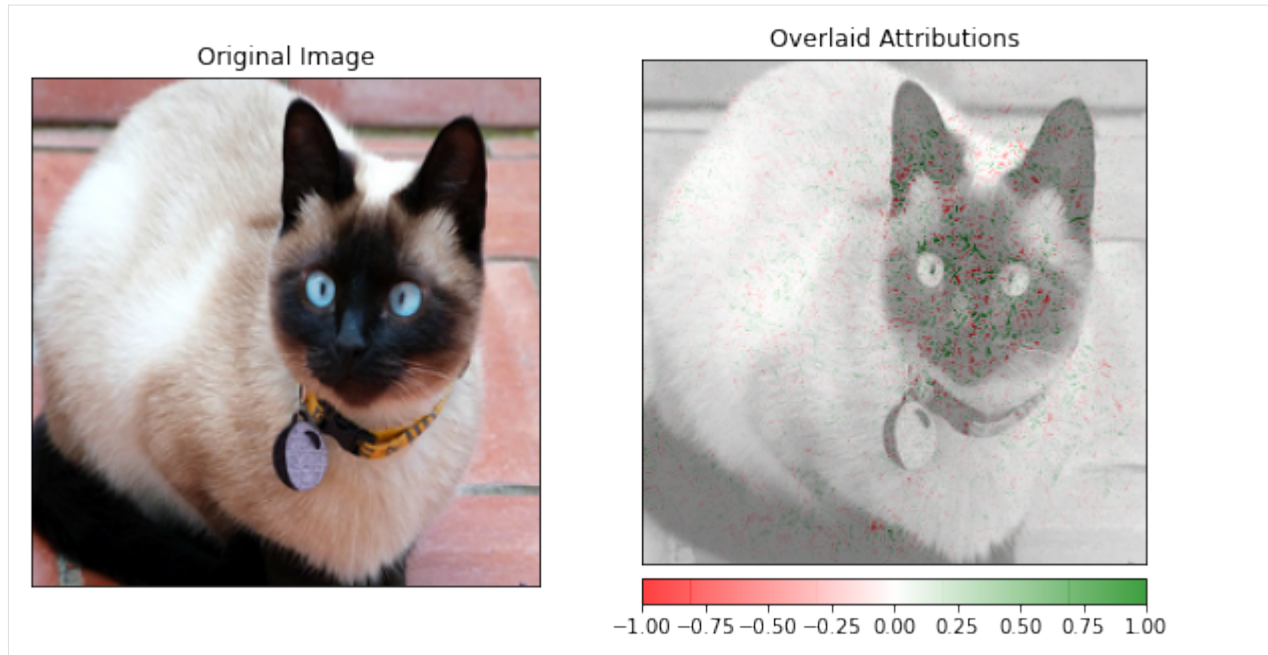
```
[13]: explanation = ig.explain(instance,
                                baselines=baselines,
                                target=predictions)
```

```
[14]: attrs = explanation.attributions[0]
```

Sample image from the test dataset and its attributions. The attributions are shown by overlaying the attributions values for each pixel to the original image. The attribution value for a pixel is obtained by summing up the attributions values for the three color channels. The attributions are scaled in a $[-1, 1]$ range: red pixel represents negative attributions, while green pixels represents positive attributions. The original image is shown in gray scale for clarity.

```
[15]: fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))
visualize_image_attr(attr=None, original_image=data[i], method='original_image',
                    title='Original Image', plt_fig_axis=(fig, ax[0]), use_pyplot=False);

visualize_image_attr(attr=attrs.squeeze(), original_image=data[i], method='blended_heat_
↪map',
                    sign='all', show_colorbar=True, title='Overlaid Attributions',
                    plt_fig_axis=(fig, ax[1]), use_pyplot=True);
```



8.8.2 Integrated gradients for text classification on the IMDB dataset

In this example, we apply the integrated gradients method to a sentiment analysis model trained on the IMDB dataset. In text classification models, integrated gradients define an attribution value for each word in the input sentence. The attributions are calculated considering the integral of the model gradients with respect to the word embedding layer along a straight path from a baseline instance x' to the input instance x . A description of the method can be found [here](#). Integrated gradients was originally proposed in Sundararajan et al., “[Axiomatic Attribution for Deep Networks](#)”

The IMDB data set contains 50K movie reviews labelled as positive or negative. We train a convolutional neural network classifier with a single 1-d convolutional layer followed by a fully connected layer. The reviews in the dataset are truncated at 100 words and each word is represented by 50-dimensional word embedding vector. We calculate attributions for the elements of the embedding layer.

Note

To enable support for IntegratedGradients, you may need to run

```
pip install alibi[tensorflow]
```

```
[1]: import tensorflow as tf
import numpy as np
import os
import pandas as pd
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Embedding, Conv1D, GlobalMaxPooling1D, Dropout
from tensorflow.keras.utils import to_categorical
from alibi.explainers import IntegratedGradients
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # True
```

```
TF version: 2.6.0
Eager execution enabled: True
```

Load data

Loading the imdb dataset.

```
[2]: max_features = 10000
     maxlen = 100
```

```
[3]: print('Loading data...')
     (x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
     test_labels = y_test.copy()
     train_labels = y_train.copy()
     print(len(x_train), 'train sequences')
     print(len(x_test), 'test sequences')
     y_train, y_test = to_categorical(y_train), to_categorical(y_test)

     print('Pad sequences (samples x time)')
     x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
     x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
     print('x_train shape:', x_train.shape)
     print('x_test shape:', x_test.shape)

     index = imdb.get_word_index()
     reverse_index = {value: key for (key, value) in index.items()}

Loading data...
25000 train sequences
25000 test sequences
Pad sequences (samples x time)
x_train shape: (25000, 100)
x_test shape: (25000, 100)
```

A sample review from the test set. Note that unknown words are replaced with ‘UNK’

```
[4]: def decode_sentence(x, reverse_index):
     # the `-3` offset is due to the special tokens used by keras
     # see https://stackoverflow.com/questions/42821330/restore-original-text-from-keras-s-imdb-dataset
     return " ".join([reverse_index.get(i - 3, 'UNK') for i in x])
```

```
[5]: print(decode_sentence(x_test[1], reverse_index))
```

```
a powerful study of loneliness sexual UNK and desperation be patient UNK up the
↳ atmosphere and pay attention to the wonderfully written script br br i praise robert
↳ altman this is one of his many films that deals with unconventional fascinating
↳ subject matter this film is disturbing but it's sincere and it's sure to UNK a strong
```

(continues on next page)

(continued from previous page)

```

↪emotional response from the viewer if you want to see an unusual film some might even.
↪say bizarre this is worth the time br br unfortunately it's very difficult to find in.
↪video stores you may have to buy it off the internet

```

Train Model

The model includes one convolutional layer and reaches a test accuracy of 0.85. If `save_model = True`, a local folder `./model_imdb` will be created and the trained model will be saved in that folder. If the model was previously saved, it can be loaded by setting `load_model = True`.

```

[6]: batch_size = 32
      embedding_dims = 50
      filters = 250
      kernel_size = 3
      hidden_dims = 250

[7]: load_model = False
      save_model = True

[8]: filepath = './model_imdb/' # change to directory where model is downloaded
      if load_model:
          model = tf.keras.models.load_model(os.path.join(filepath, 'model.h5'))
      else:
          print('Build model...')

          inputs = Input(shape=(maxlen,), dtype=tf.int32)
          embedded_sequences = Embedding(max_features,
                                         embedding_dims)(inputs)

          out = Conv1D(filters,
                       kernel_size,
                       padding='valid',
                       activation='relu',
                       strides=1)(embedded_sequences)
          out = Dropout(0.4)(out)
          out = GlobalMaxPooling1D()(out)
          out = Dense(hidden_dims,
                      activation='relu')(out)
          out = Dropout(0.4)(out)
          outputs = Dense(2, activation='softmax')(out)

          model = Model(inputs=inputs, outputs=outputs)
          model.compile(loss='categorical_crossentropy',
                       optimizer='adam',
                       metrics=['accuracy'])

          print('Train...')
          model.fit(x_train, y_train,
                   batch_size=256,
                   epochs=3,
                   validation_data=(x_test, y_test))

```

(continues on next page)

(continued from previous page)

```

if save_model:
    if not os.path.exists(filepath):
        os.makedirs(filepath)
        model.save(os.path.join(filepath, 'model.h5'))

```

Build model...

Train...

Epoch 1/3

98/98 [=====] - 13s 130ms/step - loss: 0.6030 - accuracy: 0.
 ↳6534 - val_loss: 0.4228 - val_accuracy: 0.8192

Epoch 2/3

98/98 [=====] - 14s 146ms/step - loss: 0.3223 - accuracy: 0.
 ↳8631 - val_loss: 0.3489 - val_accuracy: 0.8542

Epoch 3/3

98/98 [=====] - 19s 197ms/step - loss: 0.2128 - accuracy: 0.
 ↳9177 - val_loss: 0.3327 - val_accuracy: 0.8545

Calculate integrated gradients

The integrated gradients attributions are calculated with respect to the embedding layer for 10 samples from the test set. Since the model uses a word to vector embedding with vector dimensionality of 50 and sequence length of 100 words, the dimensionality of the attributions is (10, 100, 50). In order to obtain a single attribution value for each word, we sum all the attribution values for the 50 elements of each word's vector representation.

The default baseline is used in this example which is internally defined as a sequence of zeros. In this case, this corresponds to a sequence of padding characters (**NB:** in general the numerical value corresponding to a “non-informative” baseline such as the PAD token will depend on the tokenizer used, make sure that the numerical value of the baseline used corresponds to your desired token value to avoid surprises). The path integral is defined as a straight line from the baseline to the input image. The path is approximated by choosing 50 discrete steps according to the Gauss-Legendre method.

```

[9]: layer = model.layers[1]
     layer

```

```

[9]: <keras.layers.embeddings.Embedding at 0x7fb80bdf1e50>

```

```

[10]: n_steps = 50
      method = "gausslegendre"
      internal_batch_size = 100
      nb_samples = 10
      ig = IntegratedGradients(model,
                               layer=layer,
                               n_steps=n_steps,
                               method=method,
                               internal_batch_size=internal_batch_size)

```

```

[11]: x_test_sample = x_test[:nb_samples]
      predictions = model(x_test_sample).numpy().argmax(axis=1)
      explanation = ig.explain(x_test_sample,
                              baselines=None,
                              target=predictions,
                              attribute_to_layer_inputs=False)

```

```
[12]: # Metadata from the explanation object
      explanation.meta
```

```
[12]: {'name': 'IntegratedGradients',
      'type': ['whitebox'],
      'explanations': ['local'],
      'params': {'method': 'gausslegendre',
                  'n_steps': 50,
                  'internal_batch_size': 100,
                  'layer': 1}}
```

```
[13]: # Data fields from the explanation object
      explanation.data.keys()
```

```
[13]: dict_keys(['attributions', 'X', 'forward_kwargs', 'baselines', 'predictions', 'deltas',
      ↪ 'target'])
```

```
[14]: # Get attributions values from the explanation object
```

```
      attrs = explanation.attributions[0]
      print('Attributions shape:', attrs.shape)
```

```
Attributions shape: (10, 100, 50)
```

Sum attributions

```
[15]: attrs = attrs.sum(axis=2)
      print('Attributions shape:', attrs.shape)
```

```
Attributions shape: (10, 100)
```

Visualize attributions

```
[16]: i = 1
      x_i = x_test_sample[i]
      attrs_i = attrs[i]
      pred = predictions[i]
      pred_dict = {1: 'Positive review', 0: 'Negative review'}
```

```
[17]: print('Predicted label = {}: {}'.format(pred, pred_dict[pred]))
```

```
Predicted label = 1: Positive review
```

We can visualize the attributions for the text instance by mapping the values of the attributions onto a matplotlib colormap. Below we define some utility functions for doing this.

```
[18]: from IPython.display import HTML
      def hlstr(string, color='white'):
          """
          Return HTML markup highlighting text with the desired color.
          """
          return f"<mark style=background-color:{color}>{string} </mark>"
```

```
[19]: def colorize(attrs, cmap='PiYG'):
      """
      Compute hex colors based on the attributions for a single instance.
      Uses a diverging colorscale by default and normalizes and scales
      the colormap so that colors are consistent with the attributions.
      """
      import matplotlib as mpl
      cmap_bound = np.abs(attrs).max()
      norm = mpl.colors.Normalize(vmin=-cmap_bound, vmax=cmap_bound)
      cmap = mpl.cm.get_cmap(cmap)

      # now compute hex values of colors
      colors = list(map(lambda x: mpl.colors.rgb2hex(cmap(norm(x))), attrs))
      return colors
```

Below we visualize the attribution values (highlighted in the text) having the highest positive attributions. Words with high positive attribution are highlighted in shades of green and words with negative attribution in shades of pink. Stronger shading corresponds to higher attribution values. Positive attributions can be interpreted as increase in probability of the predicted class (“Positive sentiment”) while negative attributions correspond to decrease in probability of the predicted class.

```
[20]: words = decode_sentence(x_i, reverse_index).split()
      colors = colorize(attrs_i)
```

```
[21]: HTML("".join(list(map(hlstr, words, colors))))
```

```
[21]: <IPython.core.display.HTML object>
```

8.8.3 Integrated gradients for MNIST

In this notebook we apply the integrated gradients method to a convolutional network trained on the MNIST dataset. Integrated gradients defines an attribution value for each feature of the input instance (in this case for each pixel in the image) by integrating the model’s gradients with respect to the input along a straight path from a baseline instance x' to the input instance x .

A more detailed description of the method can be found [here](#). Integrated gradients was originally proposed in Sundararajan et al., “Axiomatic Attribution for Deep Networks”.

Note

To enable support for IntegratedGradients, you may need to run

```
pip install alibi[tensorflow]
```

```
[2]: import numpy as np
      import os
      import tensorflow as tf
      from tensorflow.keras.layers import Activation, Conv2D, Dense, Dropout
      from tensorflow.keras.layers import Flatten, Input, Reshape, MaxPooling2D
      from tensorflow.keras.models import Model
      from tensorflow.keras.utils import to_categorical
```

(continues on next page)

(continued from previous page)

```

from alibi.explainers import IntegratedGradients
import matplotlib.pyplot as plt
print('TF version: ', tf.__version__)
print('Eager execution enabled: ', tf.executing_eagerly()) # True

TF version: 2.5.0
Eager execution enabled: True

```

Load data

Loading and preparing the MNIST data set.

```

[3]: train, test = tf.keras.datasets.mnist.load_data()
X_train, y_train = train
X_test, y_test = test
test_labels = y_test.copy()
train_labels = y_train.copy()

X_train = X_train.reshape(-1, 28, 28, 1).astype('float64') / 255
X_test = X_test.reshape(-1, 28, 28, 1).astype('float64') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)

(60000, 28, 28, 1) (60000, 10) (10000, 28, 28, 1) (10000, 10)

```

Train model

Train a convolutional neural network on the MNIST dataset. The model includes 2 convolutional layers and it reaches a test accuracy of 0.98. If `save_model = True`, a local folder `./model_mnist` will be created and the trained model will be saved in that folder. If the model was previously saved, it can be loaded by setting `load_mnist_model = True`.

```

[4]: load_mnist_model = False
save_model = True

[5]: filepath = './model_mnist/' # change to directory where model is saved
if load_mnist_model:
    model = tf.keras.models.load_model(os.path.join(filepath, 'model.h5'))
else:
    # define model
    inputs = Input(shape=(X_train.shape[1:]), dtype=tf.float64)
    x = Conv2D(64, 2, padding='same', activation='relu')(inputs)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(.3)(x)

    x = Conv2D(32, 2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(.3)(x)

    x = Flatten()(x)

```

(continues on next page)

(continued from previous page)

```

x = Dense(256, activation='relu')(x)
x = Dropout(.5)(x)
logits = Dense(10, name='logits')(x)
outputs = Activation('softmax', name='softmax')(logits)
model = Model(inputs=inputs, outputs=outputs)
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# train model
model.fit(X_train,
        y_train,
        epochs=6,
        batch_size=256,
        verbose=1,
        validation_data=(X_test, y_test)
    )
if save_model:
    if not os.path.exists(filepath):
        os.makedirs(filepath)
    model.save(os.path.join(filepath, 'model.h5'))

```

```

Epoch 1/6
235/235 [=====] - 16s 65ms/step - loss: 0.5084 - accuracy: 0.
↪8374 - val_loss: 0.1216 - val_accuracy: 0.9625
Epoch 2/6
235/235 [=====] - 14s 60ms/step - loss: 0.1686 - accuracy: 0.
↪9488 - val_loss: 0.0719 - val_accuracy: 0.9781
Epoch 3/6
235/235 [=====] - 17s 70ms/step - loss: 0.1205 - accuracy: 0.
↪9634 - val_loss: 0.0520 - val_accuracy: 0.9841
Epoch 4/6
235/235 [=====] - 18s 76ms/step - loss: 0.0979 - accuracy: 0.
↪9702 - val_loss: 0.0443 - val_accuracy: 0.9863
Epoch 5/6
235/235 [=====] - 16s 69ms/step - loss: 0.0844 - accuracy: 0.
↪9733 - val_loss: 0.0382 - val_accuracy: 0.9872
Epoch 6/6
235/235 [=====] - 14s 59ms/step - loss: 0.0742 - accuracy: 0.
↪9768 - val_loss: 0.0364 - val_accuracy: 0.9875

```

Calculate integrated gradients

The `IntegratedGradients` class implements the integrated gradients attribution method. A description of the method can be found [here](#).

In the following example, the baselines (i.e. the starting points of the path integral) are black images (all pixel values are set to zero). This means that black areas of the image will always have zero attribution. The path integral is defined as a straight line from the baseline to the input image. The path is approximated by choosing 50 discrete steps according to the Gauss-Legendre method.

```
[6]: # Initialize IntegratedGradients instance
n_steps = 50
method = "gausslegendre"
ig = IntegratedGradients(model,
                        n_steps=n_steps,
                        method=method)
```

```
[7]: # Calculate attributions for the first 10 images in the test set
nb_samples = 10
X_test_sample = X_test[:nb_samples]
predictions = model(X_test_sample).numpy().argmax(axis=1)
explanation = ig.explain(X_test_sample,
                      baselines=None,
                      target=predictions)
```

```
[8]: # Metadata from the explanation object
explanation.meta
```

```
[8]: {'name': 'IntegratedGradients',
      'type': ['whitebox'],
      'explanations': ['local'],
      'params': {'method': 'gausslegendre',
                  'n_steps': 50,
                  'internal_batch_size': 100,
                  'layer': 0}}
```

```
[9]: # Data fields from the explanation object
explanation.data.keys()
```

```
[9]: dict_keys(['attributions', 'X', 'baselines', 'predictions', 'deltas', 'target'])
```

```
[10]: # Get attributions values from the explanation object
attrs = explanation.attributions[0]
```

Visualize attributions

Sample images from the test dataset and their attributions.

- The first column shows the original image.
- The second column shows the values of the attributions.
- The third column shows the positive valued attributions.
- The fourth column shows the negative valued attributions.

The attributions are calculated using the black image as a baseline for all samples.

```
[12]: fig, ax = plt.subplots(nrows=3, ncols=4, figsize=(10, 7))
image_ids = [0, 1, 9]
cmap_bound = np.abs(attrs[[0, 1, 9]]).max()

for row, image_id in enumerate(image_ids):
    # original images
```

(continues on next page)

(continued from previous page)

```

ax[row, 0].imshow(X_test[image_id].squeeze(), cmap='gray')
ax[row, 0].set_title(f'Prediction: {predictions[image_id]}')

# attributions
attr = attrs[image_id]
im = ax[row, 1].imshow(attr.squeeze(), vmin=-cmap_bound, vmax=cmap_bound, cmap='PiYG'
↪')

# positive attributions
attr_pos = attr.clip(0, 1)
im_pos = ax[row, 2].imshow(attr_pos.squeeze(), vmin=-cmap_bound, vmax=cmap_bound, ↪
↪cmap='PiYG')

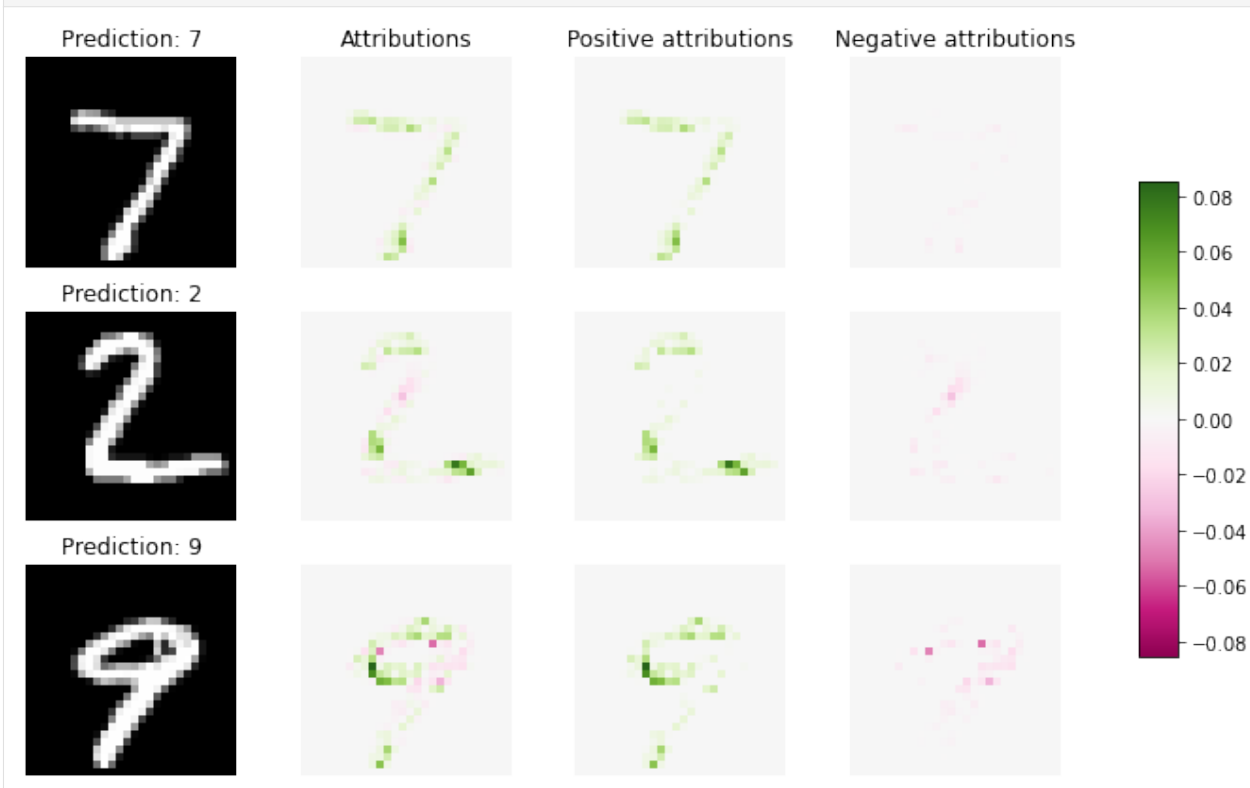
# negative attributions
attr_neg = attr.clip(-1, 0)
im_neg = ax[row, 3].imshow(attr_neg.squeeze(), vmin=-cmap_bound, vmax=cmap_bound, ↪
↪cmap='PiYG')

ax[0, 1].set_title('Attributions');
ax[0, 2].set_title('Positive attributions');
ax[0, 3].set_title('Negative attributions');

for ax in fig.axes:
    ax.axis('off')

fig.colorbar(im, cax=fig.add_axes([0.95, 0.25, 0.03, 0.5]));

```



8.8.4 Integrated gradients for transformers models

In this example, we apply the integrated gradients method to two different sentiment analysis models. The first one is a pretrained sentiment analysis model from the `transformers` library. The second model is a combination of a pretrained (distil)BERT model and a simple feed forward network. The entire model, **(distil)BERT** and feed forward network, is trained on the **IMDB reviews** dataset.

In text classification models, **integrated gradients (IG)** define an attribution value for each word in the input sentence. The attributions are calculated considering the integral of the model gradients with respect to the word embedding layer along a straight path from a baseline instance x' to the input instance x . A description of the method can be found [here](#). Integrated gradients was originally proposed in Sundararajan et al., “[Axiomatic Attribution for Deep Networks](#)”

Note

To enable support for IntegratedGradients, you may need to run

```
pip install alibi[tensorflow]
```

```
[1]: import re
import os
import numpy as np
import matplotlib as mpl
import matplotlib.cm

from tqdm import tqdm
from typing import Optional, Union, List, Dict, Tuple
from IPython.display import HTML

import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.datasets import imdb
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import SparseCategoricalCrossentropy

from transformers.optimization_tf import WarmUp
from transformers import TFAutoModelForSequenceClassification, AutoTokenizer, \
↳ PreTrainedTokenizer

from alibi.explainers import IntegratedGradients
```

Here we define some functions needed to process the data and visualize. For consistency with other *text examples* in alibi, we will use the **IMDB reviews** dataset provided by Keras. Since the dataset consists of reviews that are already tokenized, we need to decode each sentence and re-convert them into tokens using the **(distil)BERT** tokenizer.

```
[2]: def decode_sentence(x: List[int], reverse_index: Dict[int, str], unk_token: str = '[UNK]
↳ ') -> str:
    """
    Decodes the tokenized sentences from keras IMDB dataset into plain text.

    Parameters
    -----
```

(continues on next page)

(continued from previous page)

```

x
    List of integers to be decoded.
reverse_index
    Reverse index map, from `int` to `str`.
unk_token
    Unkown token to be used.

Returns
-----
    Decoded sentence.
    """
    # the `-3` offset is due to the special tokens used by keras
    # see https://stackoverflow.com/questions/42821330/restore-original-text-from-keras-s-imdb-dataset
    ↪ return " ".join([reverse_index.get(i - 3, unk_token) for i in x])

def process_sentences(sentence: List[str],
                     tokenizer: PreTrainedTokenizer,
                     max_len: int) -> Dict[str, np.ndarray]:
    """
    Tokenize the text sentences.

    Parameters
    -----
    sentence
        Sentence to be processed.
    tokenizer
        Tokenizer to be used.
    max_len
        Controls the maximum length to use by one of the truncation/padding parameters.

    Returns
    -----
    Tokenized representation containing:
    - input_ids
    - attention_mask
    """
    # since we are using the model for classification, we need to include special char_
    ↪ (i.e., '[CLS]', '[SEP]')
    # check the example here: https://huggingface.co/transformers/v4.4.2/quicktour.html
    z = tokenizer(sentence,
                  add_special_tokens=True,
                  padding='max_length',
                  max_length=max_len,
                  truncation=True,
                  return_attention_mask = True,
                  return_tensors='np')

    return z

def process_input(sentence: List[str],

```

(continues on next page)

(continued from previous page)

```

        tokenizer: PreTrainedTokenizer,
        max_len: int) -> Tuple[np.ndarray, dict]:
    """
    Preprocess input sentence befor sending to transformer model.

    Parameters
    -----
    sentence
        Sentence to be processed.
    tokenizer
        Tokenizer to be used.
    max_len
        Controls the maximum length to use by one of the truncation/padding parameters.

    Returns
    -----
    Tuple consisting of the input_ids and a dictionary contaning the attention_mask.
    """
    # tokenize the sentences using the transformer's tokenizer.
    tokenized_samples = process_sentences(sentence, tokenizer, max_len)
    X_test = tokenized_samples['input_ids'].astype(np.int32)

    # the values of the kwargs have to be `tf.Tensor`.
    # see transformers issue #14404: https://github.com/huggingface/transformers/issues/
    ↪ 14404
    # solved from v4.16.0
    kwargs = {k: tf.constant(v) for k, v in tokenized_samples.items() if k != 'input_ids
    ↪ '}
    return X_test, kwargs

```

```

[3]: def hlstr(string: str , color: str = 'white') -> str:
    """
    Return HTML markup highlighting text with the desired color.
    """
    return f"<mark style=background-color:{color}>{string} </mark>"

def colorize(attrs: np.ndarray, cmap: str = 'PiYG') -> List:
    """
    Compute hex colors based on the attributions for a single instance.
    Uses a diverging colorscale by default and normalizes and scales
    the colormap so that colors are consistent with the attributions.

    Parameters
    -----
    attrs
        Attributions to be visualized.
    cmap
        Matplotlib cmap type.
    """
    cmap_bound = np.abs(attrs).max()
    norm = mpl.colors.Normalize(vmin=-cmap_bound, vmax=cmap_bound)

```

(continues on next page)

(continued from previous page)

```

cmap = mpl.cm.get_cmap(cmap)
return list(map(lambda x: mpl.colors.rgb2hex(cmap(norm(x))), attrs))

def display(X: np.ndarray,
            attrs: np.ndarray,
            tokenizer: PreTrainedTokenizer,
            pred: np.ndarray) -> None:
    """
    Display the attribution of a given instance.

    Parameters
    -----
    X
        Instance to display the attributions for.
    attrs
        Attributions values for the given instance.
    tokenizer
        Tokenizer to be used for decoding.
    pred
        Classification label (prediction) for the given instance.
    """
    pred_dict = {1: 'Positive review', 0: 'Negative review'}

    # remove padding
    fst_pad_indices = np.where(X == tokenizer.pad_token_id)[0]
    if len(fst_pad_indices) > 0:
        X, attrs = X[:fst_pad_indices[0]], attrs[:fst_pad_indices[0]]

    # decode tokens and get colors
    tokens = [tokenizer.decode([X[i]]) for i in range(len(X))]
    colors = colorize(attrs)

    print(f'Predicted label = {pred}: {pred_dict[pred]}')
    return HTML("".join(list(map(hlstr, tokens, colors))))

```

Automodel

In this section, we will use the Tensorflow auto model for sequence classification provided by the [transformers](#) library.

The model is pretrained on the [Stanford Sentiment Treebank \(SST\)](#) dataset. The **Stanford Sentiment Treebank** is the first corpus with fully labeled parse trees that allows for a complete analysis of the compositional effects of sentiment in language.

Each phrase is labeled as either negative, somewhat negative, neutral, somewhat positive or positive. The corpus with all 5 labels is referred to as **SST-5** or **SST fine-grained**. Binary classification experiments on full sentences (negative or somewhat negative vs somewhat positive or positive with neutral sentences discarded) refer to the dataset as **SST-2** or **SST binary**. In this example, we will use a text classifier pretrained on the **SST-2** dataset.

```

[4]: # load model and tokenizer
model_name = "distilbert-base-uncased-finetuned-sst-2-english"

```

(continues on next page)

(continued from previous page)

```
auto_model_distilbert = TFAutoModelForSequenceClassification.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

All model checkpoint layers were used when initializing `TFDistilBertForSequenceClassification`.

All the layers of `TFDistilBertForSequenceClassification` were initialized from the model checkpoint at `distilbert-base-uncased-finetuned-sst-2-english`.

If your task is similar to the task the model of the checkpoint was trained on, you can already use `TFDistilBertForSequenceClassification` for predictions without further training.

The `auto_model_distilbert` output is a custom object containing the output logits. We use a wrapper to transform the output into a tensor and apply a softmax function to the logits.

```
[5]: class AutoModelWrapper(keras.Model):
    def __init__(self, transformer: keras.Model, **kwargs):
        """
        Constructor.

        Parameters
        -----
        transformer
            Transformer to be wrapped.
        """
        super().__init__()
        self.transformer = transformer

    def call(self,
            input_ids: Union[np.ndarray, tf.Tensor],
            attention_mask: Optional[Union[np.ndarray, tf.Tensor]] = None,
            training: bool = False):
        """
        Performs forward pass through the model.

        Parameters
        -----
        input_ids
            Indices of input sequence tokens in the vocabulary.
        attention_mask
            Mask to avoid performing attention on padding token indices.

        Returns
        -----
        Classification probabilities.
        """
        out = self.transformer(input_ids=input_ids, attention_mask=attention_mask,
                               training=training)
        return tf.nn.softmax(out.logits, axis=-1)

    def get_config(self):
        return {}
```

(continues on next page)

(continued from previous page)

```
@classmethod
def from_config(cls, config):
    return cls(**config)
```

```
[6]: auto_model = AutoModelWrapper(auto_model_distilbert)
```

Calculate integrated gradients

The auto model consists of a main **distilBERT** layer (layer 0) followed by two dense layers.

```
[7]: auto_model.layers[0].layers
```

```
[7]: [<transformers.models.distilbert.modeling_tf_distilbert.TFDistilBertMainLayer at 0x7f4cb5435e80>,
      <keras.layers.core.dense.Dense at 0x7f4ca8058c10>,
      <keras.layers.core.dense.Dense at 0x7f4ca8058e80>,
      <keras.layers.core.dropout.Dropout at 0x7f4ca01db1f0>]
```

We will proceed with the embedding layer from **distilBERT**. We calculate attributions to the outputs of the **embedding layer** for which we can easily construct an appropriate baseline for the **IG** by replacing the regular tokens with the [PAD] token (i.e. a neutral token) and keeping the other special tokens (e.g. [CLS], [SEP], [UNK], [PAD]). By including special tokens such as [CLS], [SEP], [UNK], we ensure that the attribution for those tokens will be 0 if we use the embedding layer. The 0 attribution is due to integration between $[x, x]$ which is 0. Note that if we considered a hidden layer instead, we would inevitably capture higher order interaction between the input tokens. Moreover, the embedding layer is our first choice since we cannot compute attributions for the raw input due to its discrete structure (i.e., we cannot differentiate the output of the model with respect to the discrete input representation). **That being said, you can use any other layer and compute attributions to the outputs of it instead.**

```
[8]: # Extracting the embeddings layer
layer = auto_model.layers[0].layers[0].embeddings

# # Extract the first layer from the transformer
# layer = auto_model.layers[0].layers[0].transformer.layer[0]
```

```
[9]: n_steps = 50
internal_batch_size = 5
method = "gausslegendre"

# define Integrated Gradients explainer
ig = IntegratedGradients(auto_model,
                        layer=layer,
                        n_steps=n_steps,
                        method=method,
                        internal_batch_size=internal_batch_size)
```

Here we consider some simple sentences such as “I love you, I like you”, “I love you, I like you, but I also kind of dislike you”.

```
[10]: # define some text to be explained
text_samples = [
    'I love you, I like you',
```

(continues on next page)

(continued from previous page)

```

    'I love you, I like you, but I also kind of dislike you',
    'Everything is so nice about you'
]

# process input to be explained
X_test, kwargs = process_input(sentence=text_samples,
                               tokenizer=tokenizer,
                               max_len=256)

```

```

[11]: # get predictions
      predictions = auto_model(X_test, **kwargs).numpy().argmax(axis=1)

      # get the baseline
      mask = np.isin(X_test, tokenizer.all_special_ids)
      baselines = X_test * mask + tokenizer.pad_token_id * (1 - mask)

      # get explanation
      explanation = ig.explain(X_test,
                              forward_kwargs=kwargs,
                              baselines=baselines,
                              target=predictions)

```

Let's check the attributions' shapes.

```

[12]: # Get attributions values from the explanation object
      attrs = explanation.attributions[0]
      print('Attributions shape:', attrs.shape)

      Attributions shape: (3, 256, 768)

```

As you can see, the attribution of each token corresponds to a tensor of 768 elements. We compress all this information into a single number by summing up all 768 components. The nice thing about this is that we still remain consistent with the **Completeness Axiom**, which states that the attributions add up to the difference between the output of our model for the given instance and the output of our model for the given baseline.

```

[13]: attrs = attrs.sum(axis=2)
      print('Attributions shape:', attrs.shape)

      Attributions shape: (3, 256)

```

```

[14]: index = 1
      display(X=X_test[index], attrs=attrs[index], pred=predictions[index],
      ↪tokenizer=tokenizer)

```

Predicted label = 0: Negative review

```

[14]: <IPython.core.display.HTML object>

```

Note that since the sentence is classified as negative, words like ``dislike`` contribute positively to the score while words like ``love`` contribute negatively.

Sentiment analysis on IMDB with fine-tuned model head.

Load and process data

```
[15]: # constants
max_features = 10000

# load imdb reviews datasets.
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)

# remove the first integer token which is a special character that marks the beginning
# of the sentence
x_train = [x[1:] for x in x_train]
x_test = [x[1:] for x in x_test]

# get mappings. The keys are transformed to lower case since we will use uncased models.
reverse_index = {value: key.lower() for (key, value) in imdb.get_word_index().items()}
```

Load model and corresponding tokenizer

Now we have to load the model and the corresponding tokenizer. You can chose between the **BERT** model or the **distilBERT** model. Note that we will be finetuning those models which will require access to a **GPU**. In our experiments, we trained distilBERT on a single **Quadro RTX 5000** which requires around **5GB** of memory. The entire training took around **5-6 min**. We recommend using **distilBERT** as it is lighter and we did not noticed a big difference in performance between the two models after finetuning.

```
[16]: # choose whether to use the BERT or distilBERT model by selecting the appropriate name
model_name = 'distilbert-base-uncased'
# model_name = 'bert-base-uncased'
```

```
[17]: # load model and tokenizer
model = TFAutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# define maximum input length
max_len = 256

if model_name == 'bert-base-uncased':
    # training parameters: https://huggingface.co/fabriceyh/bert-base-uncased-imdb
    init_lr = 5e-05
    min_lr_ratio = 0
    batch_size = 8
    num_warmup_steps = 1546
    num_train_steps = 15468
    power = 1.0

elif model_name == 'distilbert-base-uncased':
    # training parameters: https://huggingface.co/lvwerra/distilbert-imdb
    init_lr = 5e-05
    min_lr_ratio = 0
    batch_size = 16
```

(continues on next page)

(continued from previous page)

```

num_warmup_steps = 0
num_train_steps = int(np.ceil(len(x_train) / batch_size))
power = 1.0

```

```

else:
    raise ValueError('Unknown model name.')

```

Some layers from the model checkpoint at distilbert-base-uncased were not used when
↳ initializing TFDistilBertForSequenceClassification: ['vocab_transform', 'vocab_
↳ projector', 'activation_13', 'vocab_layer_norm']
- This IS expected if you are initializing TFDistilBertForSequenceClassification from
↳ the checkpoint of a model trained on another task or with another architecture (e.g.
↳ initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing TFDistilBertForSequenceClassification
↳ from the checkpoint of a model that you expect to be exactly identical (initializing a
↳ BertForSequenceClassification model from a BertForSequenceClassification model).
Some layers of TFDistilBertForSequenceClassification were not initialized from the model
↳ checkpoint at distilbert-base-uncased and are newly initialized: ['dropout_39', 'pre_
↳ classifier', 'classifier']
You should probably TRAIN this model on a down-stream task to be able to use it for
↳ predictions and inference.

Decoding each sentence in the **Keras IMDB** tokenized dataset to obtain the corresponding plain text. The dataset is already in a pretty good shape, so we don't need to do extra preprocessing. The only thing that we do is to replace the unknown tokens with the appropriate tokenizer's unknown token (i.e., [UNK])

```

[18]: X_train, X_test = [], []

# decode training sentences
for i in range(len(x_train)):
    tr_sentence = decode_sentence(x_train[i], reverse_index, unk_token=tokenizer.unk_
↳ token)
    X_train.append(tr_sentence)

# decode testing sentences
for i in range(len(x_test)):
    te_sentence = decode_sentence(x_test[i], reverse_index, unk_token=tokenizer.unk_
↳ token)
    X_test.append(te_sentence)

```

Retokenizing the plain text using the **(distil)BERT** tokenizer.

```

[19]: # tokenize datasets
X_train = process_sentences(X_train, tokenizer, max_len)
X_test = process_sentences(X_test, tokenizer, max_len)

```

Construct the Tensorflow datasets for training and testing.

```

[20]: train_ds = tf.data.Dataset.from_tensor_slices((*X_train.values(),), y_train))
train_ds = train_ds.shuffle(1024).batch(batch_size).repeat()

test_ds = tf.data.Dataset.from_tensor_slices((*X_test.values(),), y_test))
test_ds = test_ds.batch(batch_size)

```

Train model

Here we train a classification model by leveraging the pretrained **(distil)BERT** transformer.

```
[21]: filepath = './model_transformers/' # change to desired save directory
checkpoint_path = os.path.join(filepath, model_name)
load_model = False

# define linear learning schedules
lr_schedule = tf.keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=init_lr,
    decay_steps=num_train_steps - num_warmup_steps,
    end_learning_rate=init_lr * min_lr_ratio,
    power=power,
)

# include learning rate warmup
if num_warmup_steps:
    lr_schedule = WarmUp(
        initial_learning_rate=init_lr,
        decay_schedule_fn=lr_schedule,
        warmup_steps=num_warmup_steps,
    )

if not load_model:
    # compile the model
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=lr_schedule, beta_1=0.9, beta_
→2=0.999, epsilon=1e-08),
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=tf.metrics.SparseCategoricalAccuracy(),
    )

    # fit and save the model
    model.fit(x=train_ds, validation_data=test_ds, steps_per_epoch=num_train_steps)
    model.save_pretrained(checkpoint_path)
else:
    # load and compile the model
    model = model.from_pretrained(checkpoint_path)
    model.compile(
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=tf.metrics.SparseCategoricalAccuracy(),
    )

    # evaluate the model
    model.evaluate(test_ds)

1563/1563 [=====] - 459s 291ms/step - loss: 0.2721 - sparse_
→categorical_accuracy: 0.8859 - val_loss: 0.2141 - val_sparse_categorical_accuracy: 0.
→9142
```

```
[22]: # wrap the finetuned model
auto_model = AutoModelWrapper(model)
```

Calculate integrated gradients

We pick the first 10 sentences from the test set as examples. You can easily add some of your text here too, as we exemplify it.

```
[23]: # include IMDB reviews from the test dataset
text_samples = [decode_sentence(x_test[i], reverse_index, unk_token=tokenizer.unk_token),
↳ for i in range(10)]

# include your text here
text_samples.append("best movie i've ever seen nothing bad to say about it")

# process input before passing it to the explainer
X_test, kwargs = process_input(sentence=text_samples,
                               tokenizer=tokenizer,
                               max_len=max_len)
```

We calculate the attributions with respect to the first embedding layer of the **(distil)BERT**. You can choose any other layer.

```
[24]: if model_name == 'bert-base-uncased':
    layer = auto_model.layers[0].layers[0].embeddings
    # layer = auto_model.layers[0].layers[0].encoder.layer[2]

elif model_name == 'distilbert-base-uncased':
    layer = auto_model.layers[0].layers[0].embeddings
    # layer = auto_model.layers[0].layers[0].transformer.layer[0]

else:
    raise ValueError('Unknown model name.')
```

```
[25]: n_steps = 50
method = "gausslegendre"
internal_batch_size = 5

# define Integrated Gradients explainer
ig = IntegratedGradients(auto_model,
                        layer=layer,
                        n_steps=n_steps,
                        method=method,
                        internal_batch_size=internal_batch_size)
```

```
[26]: # compute model's prediction and construct baselines
predictions = auto_model(X_test, **kwargs).numpy().argmax(axis=1)

# construct the baseline as before
mask = np.isin(X_test, tokenizer.all_special_ids)
baselines = X_test * mask + tokenizer.pad_token_id * (1 - mask)

# get explanation
explanation = ig.explain(X_test,
                      forward_kwargs=kwargs,
                      baselines=baselines,
```

(continues on next page)

(continued from previous page)

```
target=predictions)
```

```
[27]: # Get attributions values from the explanation object
```

```
attrs = explanation.attributions[0]
print('Attributions shape:', attrs.shape)
```

```
Attributions shape: (11, 256, 768)
```

```
[28]: attrs = attrs.sum(axis=2)
```

```
print('Attributions shape:', attrs.shape)
```

```
Attributions shape: (11, 256)
```

Check attributions for our example

```
[29]: index = -1
```

```
display(X=X_test[index], attrs=attrs[index], pred=predictions[index],
        tokenizer=tokenizer)
```

```
Predicted label = 1: Positive review
```

```
[29]: <IPython.core.display.HTML object>
```

Check attribution for some test examples

```
[30]: index = 0
```

```
display(X=X_test[index], attrs=attrs[index], pred=predictions[index],
        tokenizer=tokenizer)
```

```
Predicted label = 0: Negative review
```

```
[30]: <IPython.core.display.HTML object>
```

```
[31]: index = 1
```

```
display(X=X_test[index], attrs=attrs[index], pred=predictions[index],
        tokenizer=tokenizer)
```

```
Predicted label = 1: Positive review
```

```
[31]: <IPython.core.display.HTML object>
```

8.9 Kernel SHAP

8.9.1 Distributed KernelSHAP

Note

To enable SHAP support, you may need to run

```
pip install alibi[shap]
```

Introduction

In this example, KernelSHAP is used to explain a batch of instances on multiple cores. To run this example, please run `pip install alibi[ray]` first.

Warning

Windows support for the ray Python library is *in beta*. Using KernelShap in parallel is not currently supported on Windows platforms.

```
[ ]: # shap.summary_plot currently doesn't work with matplotlib>=3.6.0,
# see bug report: https://github.com/slundberg/shap/issues/2687
!pip install matplotlib==3.5.3
```

```
[ ]: import pprint
import shap
import ray
shap.initjs()

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from alibi.explainers import KernelShap
from alibi.datasets import fetch_adult
from collections import defaultdict
from scipy.special import logit
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from timeit import default_timer as timer
from typing import Dict, List, Tuple
```

Data preparation

Load and split

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of categorical variables to numbers.

```
[2]: adult = fetch_adult()
      adult.keys()

[2]: dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])
```

```
[3]: data = adult.data
      target = adult.target
      target_names = adult.target_names
      feature_names = adult.feature_names
      category_map = adult.category_map
```

Note that for your own datasets you can use our utility function `gen_category_map` to create the category map.

```
[4]: from alibi.utils import gen_category_map

[5]: np.random.seed(0)
      data_perm = np.random.permutation(np.c_[data, target])
      data = data_perm[:, :-1]
      target = data_perm[:, -1]

[6]: idx = 30000
      X_train, y_train = data[:idx, :], target[:idx]
      X_test, y_test = data[idx+1:, :], target[idx+1:]
```

Create feature transformation pipeline

Create feature pre-processor. Needs to have ‘fit’ and ‘transform’ methods. Different types of pre-processing can be applied to all or part of the features. In the example below we will standardize ordinal features and apply one-hot-encoding to categorical features.

Ordinal features:

```
[7]: ordinal_features = [x for x in range(len(feature_names)) if x not in list(category_map.
      ↪keys())]
      ordinal_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
      ↪('scaler', StandardScaler())])
```

Categorical features:

```
[8]: categorical_features = list(category_map.keys())
      categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
      ↪('onehot', OneHotEncoder(drop='first', handle_
      ↪unknown='error'))])
```

Note that in order to be able to interpret the coefficients corresponding to the categorical features, the option `drop='first'` has been passed to the `OneHotEncoder`. This means that for a categorical variable with `n` levels,

the length of the code will be $n-1$. This is necessary in order to avoid introducing feature multicollinearity, which would skew the interpretation of the results. For more information about the issue about multicollinearity in the context of linear modelling see [1].

Combine and fit:

```
[9]: preprocessor = ColumnTransformer(transformers=[('num', ordinal_transformer, ordinal_
    ↪ features),
                                           ('cat', categorical_transformer,
    ↪ categorical_features)])
preprocessor.fit(X_train)

[9]: ColumnTransformer(transformers=[('num',
    Pipeline(steps=[('imputer',
                     SimpleImputer(strategy='median')),
                     ('scaler', StandardScaler())]),
    [0, 8, 9, 10]),
    ('cat',
    Pipeline(steps=[('imputer',
                     SimpleImputer(strategy='median')),
                     ('onehot',
                     OneHotEncoder(drop='first'))]),
    [1, 2, 3, 4, 5, 6, 7, 11])])
```

Preprocess the data

```
[10]: X_train_proc = preprocessor.transform(X_train)
X_test_proc = preprocessor.transform(X_test)
```

Applying the sklearn processing pipeline modifies the column order of the original dataset. The new feature ordering is necessary in order to correctly plot visualisations, and is inferred from the preprocessor object below:

```
[11]: numerical_feats_idx = preprocessor.transformers_[0][2]
categorical_feats_idx = preprocessor.transformers_[1][2]
scaler = preprocessor.transformers_[0][1].named_steps['scaler']
num_feats_names = [feature_names[i] for i in numerical_feats_idx]
cat_feats_names = [feature_names[i] for i in categorical_feats_idx]
perm_feat_names = num_feats_names + cat_feats_names
```

```
[12]: pp = pprint.PrettyPrinter()
print("Original order:")
pp.pprint(feature_names)
print("")
print("New features order:")
pp.pprint(perm_feat_names)
```

```
Original order:
['Age',
 'Workclass',
 'Education',
 'Marital Status',
 'Occupation',
 'Relationship',
```

(continues on next page)

(continued from previous page)

```
'Race',
'Sex',
'Capital Gain',
'Capital Loss',
'Hours per week',
'Country']
```

New features order:

```
['Age',
'Capital Gain',
'Capital Loss',
'Hours per week',
'Workclass',
'Education',
'Marital Status',
'Occupation',
'Relationship',
'Race',
'Sex',
'Country']
```

Create a utility to reorder the columns of an input array so that the features have the same ordering as that induced by the preprocessor.

```
[13]: def permute_columns(X: np.ndarray, feat_names: List[str], perm_feat_names: List[str]) ->
      np.ndarray:
      """
      Permutes the original dataset so that its columns (ordered according to feat_names)
      have the order
      of the variables after transformation with the sklearn preprocessing pipeline (perm_
      feat_names).
      """

      perm_X = np.zeros_like(X)
      perm = []
      for i, feat_name in enumerate(perm_feat_names):
          feat_idx = feat_names.index(feat_name)
          perm_X[:, i] = X[:, feat_idx]
          perm.append(feat_idx)
      return perm_X, perm
```

The categorical variables will be grouped to reduce shap values variance, as shown in *this* example. To do so, the dimensionality of each categorical variable is extracted from the preprocessor:

```
[14]: # get feature names for the encoded categorical features
ohe = preprocessor.transformers_[1][1].named_steps['onehot']
fts = [feature_names[x] for x in categorical_features]
cat_enc_feat_names = ohe.get_feature_names_out(fts)
# compute encoded dimension; -1 as ohe is setup with drop='first'
feat_enc_dim = [len(cat_enc) - 1 for cat_enc in ohe.categories_]
d = {'feature_names': fts, 'encoded_dim': feat_enc_dim}
df = pd.DataFrame(data=d)
```

(continues on next page)

(continued from previous page)

```
print(df)
total_dim = df['encoded_dim'].sum()
print("The dimensionality of the encoded categorical features is {}".format(total_dim))
assert total_dim == len(cat_enc_feat_names)
```

	feature_names	encoded_dim
0	Workclass	8
1	Education	6
2	Marital Status	3
3	Occupation	8
4	Relationship	5
5	Race	4
6	Sex	1
7	Country	10

The dimensionality of the encoded categorical features is 45.

Select a subset of test instances to explain

```
[15]: def split_set(X, y, fraction, random_state=0):
      """
      Given a set X, associated labels y, splits a fraction y from X.
      """
      _, X_split, _, y_split = train_test_split(X,
                                                y,
                                                test_size=fraction,
                                                random_state=random_state,
                                                )
      print("Number of records: {}".format(X_split.shape[0]))
      print("Number of class 0: {}".format(0, len(y_split) - y_split.sum()))
      print("Number of class 1: {}".format(1, y_split.sum()))

      return X_split, y_split
```

```
[16]: fraction_explained = 0.05
X_explain, y_explain = split_set(X_test,
                                y_test,
                                fraction_explained,
                                )
X_explain_proc = preprocessor.transform(X_explain)

Number of records: 128
Number of class 0: 96
Number of class 1: 32
```

Create a version of the dataset to be explained that has the same feature ordering as that of the feature matrix after applying the preprocessing (for plotting purposes).

```
[17]: perm_X_explain, _ = permute_columns(X_explain, feature_names, perm_feat_names)
```

Fit a binary logistic regression classifier to the Adult dataset

Training

```
[18]: classifier = LogisticRegression(multi_class='multinomial',
                                     random_state=0,
                                     max_iter=500,
                                     verbose=0,
                                     )
classifier.fit(X_train_proc, y_train)

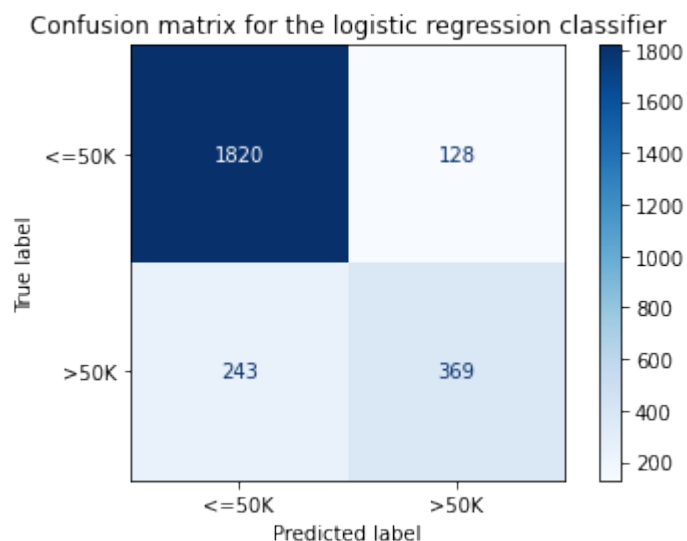
[18]: LogisticRegression(max_iter=500, multi_class='multinomial', random_state=0)
```

Model assessment

```
[19]: y_pred = classifier.predict(X_test_proc)

[20]: cm = confusion_matrix(y_test, y_pred)

[21]: title = 'Confusion matrix for the logistic regression classifier'
disp = ConfusionMatrixDisplay.from_estimator(classifier,
                                             X_test_proc,
                                             y_test,
                                             display_labels=target_names,
                                             cmap=plt.cm.Blues,
                                             normalize=None,
                                             )
disp.ax_.set_title(title);
```



```
[22]: print('Test accuracy: ', accuracy_score(y_test, classifier.predict(X_test_proc)))

Test accuracy:  0.855078125
```

Running KernelSHAP in sequential mode

A background dataset is selected.

```
[23]: start_example_idx = 0
      stop_example_idx = 100
      background_data = slice(start_example_idx, stop_example_idx)
```

Groups are specified by creating a list where each sublist contains the column indices that a given variable occupies in the preprocessed feature matrix.

```
[24]: def make_groups(num_feats_names: List[str], cat_feats_names: List[str], feat_enc_dim:
      ↪List[int]) -> Tuple[List[str], List[List[int]]]:
      """
      Given a list with numerical feat. names, categorical feat. names
      and a list specifying the lengths of the encoding for each cat.
      variable, the function outputs a list of group names, and a list
      of the same len where each entry represents the column indices that
      the corresponding categorical feature
      """

      group_names = num_feats_names + cat_feats_names
      groups = []
      cat_var_idx = 0

      for name in group_names:
          if name in num_feats_names:
              groups.append(list(range(len(groups), len(groups) + 1)))
          else:
              start_idx = groups[-1][-1] + 1 if groups else 0
              groups.append(list(range(start_idx, start_idx + feat_enc_dim[cat_var_idx] )))
              cat_var_idx += 1

      return group_names, groups

def sparse2ndarray(mat, examples=None):
    """
    Converts a scipy.sparse.csr.csr_matrix to a numpy.ndarray.
    If specified, examples is slice object specifying which selects a
    number of rows from mat and converts only the respective slice.
    """

    if examples:
        return mat[examples, :].toarray()

    return mat.toarray()

X_train_proc_d = sparse2ndarray(X_train_proc, examples=background_data)
group_names, groups = make_groups(num_feats_names, cat_feats_names, feat_enc_dim)
```

Initialise and run the explainer sequentially.

```
[25]: pred_fcn = classifier.predict_proba
      seq_lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names)
```

(continues on next page)

(continued from previous page)

```
seq_lr_explainer.fit(X_train_proc_d[background_data, :], group_names=group_names,
↳ groups=groups)
```

```
[25]: KernelShap(meta={
    'name': 'KernelShap',
    'type': ['blackbox'],
    'task': 'classification',
    'explanations': ['local', 'global'],
    'params': {
        'link': 'logit',
        'group_names': ['Age', 'Capital Gain', 'Capital Loss', 'Hours per week',
↳ 'Workclass', 'Education', 'Marital Status', 'Occupation', 'Relationship', 'Race', 'Sex
↳ ', 'Country'],
        'grouped': True,
        'groups': [[0], [1], [2], [3], [4, 5, 6, 7, 8, 9, 10, 11], [12, 13, 14, 15,
↳ 16, 17], [18, 19, 20], [21, 22, 23, 24, 25, 26, 27, 28], [29, 30, 31, 32, 33], [34,
↳ 35, 36, 37], [38], [39, 40, 41, 42, 43, 44, 45, 46, 47, 48]],
        'weights': None,
        'summarise_background': False,
        'summarise_result': None,
        'transpose': False,
        'kwargs': {}

    'version': '0.7.1dev'}
)
```

```
[26]: n_runs = 3
```

```
[27]: s_explanations, s_times = [], []
```

```
[ ]: for run in range(n_runs):
    t_start = timer()
    explanation = seq_lr_explainer.explain(sparse2ndarray(X_explain_proc))
    t_elapsed = timer() - t_start
    s_times.append(t_elapsed)
    s_explanations.append(explanation.shap_values)
```

Running KernelSHAP in distributed mode

The only change needed to distribute the computation is to pass a dictionary containing the number of (physical) CPUs available to distribute the computation to the KernelShap constructor:

```
[29]: def distrib_opts_factory(n_cpus: int) -> Dict[str, int]:
    return {'n_cpus': n_cpus}
```

```
[30]: cpu_range = range(2, 5)
distrib_avg_times = dict(zip(cpu_range, [0.0]*len(cpu_range)))
distrib_min_times = dict(zip(cpu_range, [0.0]*len(cpu_range)))
distrib_max_times = dict(zip(cpu_range, [0.0]*len(cpu_range)))
d_explanations = defaultdict(list)
```

```
[ ]: for n_cpu in cpu_range:
    opts = distrib_opts_factory(n_cpu)
    distrib_lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_
    ↪names, distributed_opts=opts)
    distrib_lr_explainer.fit(X_train_proc_d[background_data, :], group_names=group_names,
    ↪groups=groups)
    raw_times = []
    for _ in range(n_runs):
        t_start = timer()
        d_explanations[n_cpu].append(distrib_lr_explainer.explain(sparse2ndarray(X_
    ↪explain_proc), silent=True).shap_values)
        t_elapsed = timer() - t_start
        raw_times.append(t_elapsed)
    distrib_avg_times[n_cpu] = np.round(np.mean(raw_times), 3)
    distrib_min_times[n_cpu] = np.round(np.min(raw_times), 3)
    distrib_max_times[n_cpu] = np.round(np.max(raw_times), 3)
    ray.shutdown()
```

Results analysis

Timing

```
[41]: print(f"Distributed average times for {n_runs} runs (n_cpus: avg_time):")
    print(distrib_avg_times)
    print("")
    print(f"Sequential average time for {n_runs} runs:")
    print(np.round(np.mean(s_times), 3), "s")
```

```
Distributed average times for 3 runs (n_cpus: avg_time):
{2: 57.197, 3: 41.728, 4: 36.751}
```

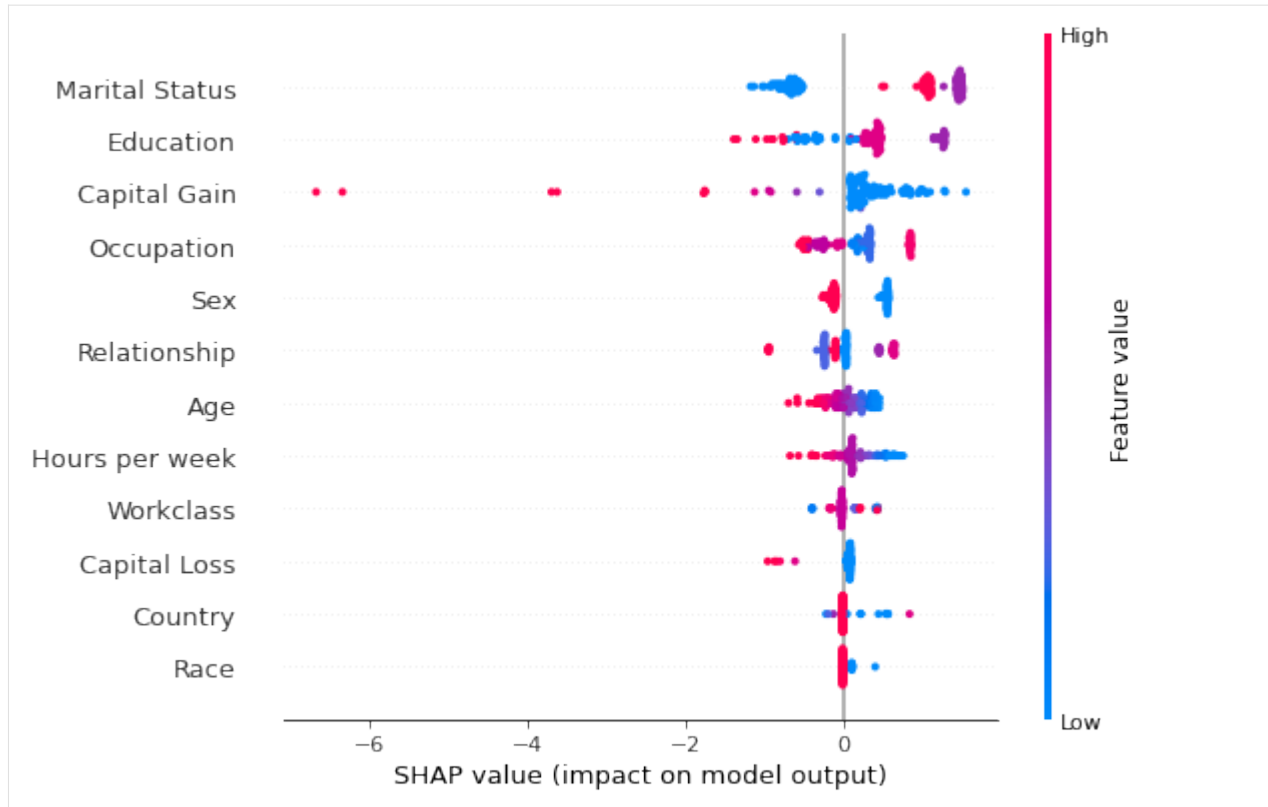
```
Sequential average time for 3 runs:
119.656 s
```

Running KernelSHAP in a distributed fashion improves the runtime as the results above show. However, the results above should not be interpreted as performance measurements since they were not run in a controlled environment. See our [blog post](#) for a more thorough analysis.

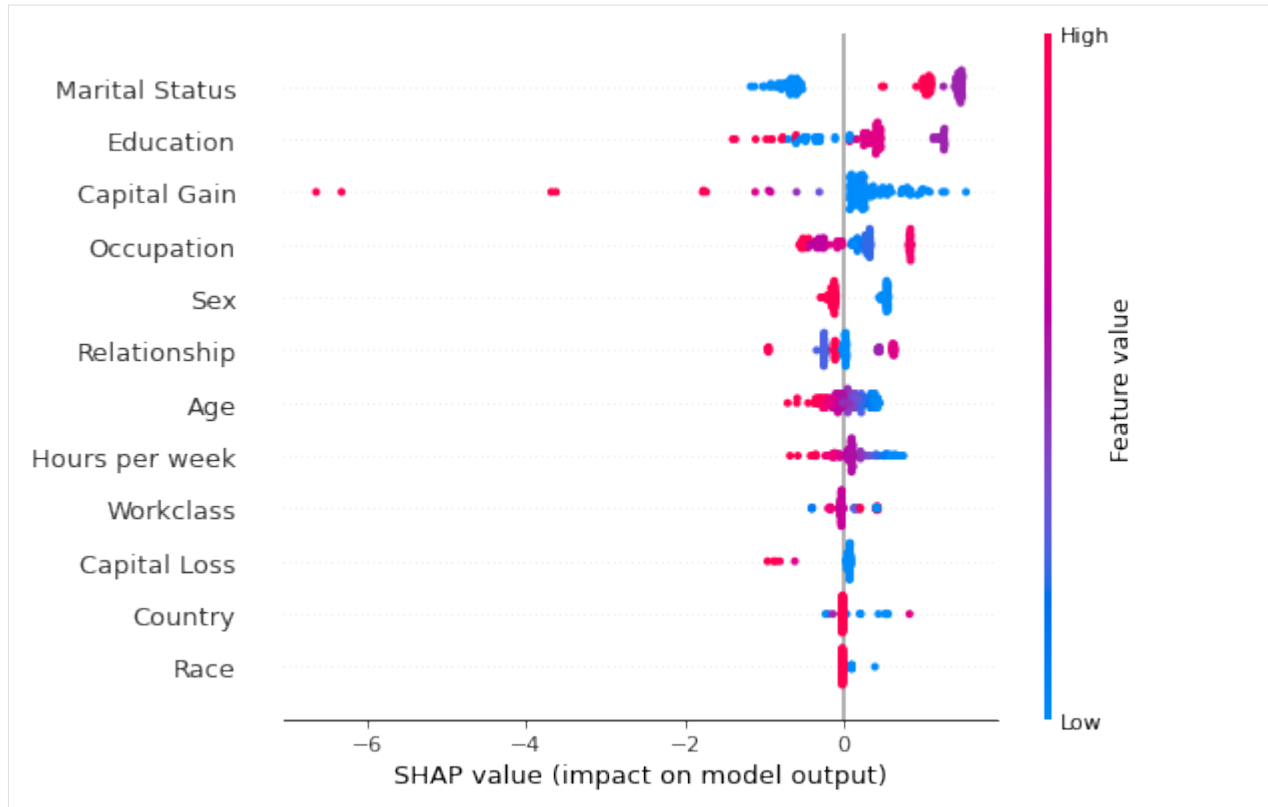
Explanations comparison

```
[33]: cls = 0 # class of prediction explained
    run = 1 # which run to compare the result for
```

```
[34]: # sequential
    shap.summary_plot(s_explanations[run][cls], perm_X_explain, perm_feat_names)
```



```
[36]: # distributed
n_cpu = 3
shap.summary_plot(d_explanations[n_cpu][run][cls], perm_X_explain, perm_feat_names)
```



Comparing the results above one sees that the running the algorithm across multiple cores gave identical results, indicating its correctness.

Conclusion

This example showed that batches of explanations can be explained much faster by simply passing `distributed_opts={'n_cpus': k}` to the `KernelShap` constructor (here `k` is the number of physical cores available). The significant runtime reduction makes it possible to explain larger datasets faster and combine shap values estimated with KernelSHAP into global explanations or use larger background datasets.

8.9.2 KernelSHAP: combining preprocessor and predictor

Note

To enable SHAP support, you may need to run

```
pip install alibi[shap]
```

Introduction

In *this* example, we showed that the categorical variables can be handled by fitting the explainer on preprocessed data and passing preprocessed data to the `explain` call. To handle the categorical variables, we either group them explicitly or sum the estimated shap values for each encoded shap dimension. An alternative way is to define our black-box model to include the preprocessor, as shown in *this* example. We now show that these methods give the same results.

```
[ ]: import shap
shap.initjs()

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from alibi.explainers import KernelShap
from alibi.datasets import fetch_adult
from scipy.special import logit
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

Data preparation

Load and split

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of categorical variables to numbers.

```
[2]: adult = fetch_adult()
adult.keys()

[2]: dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])

[3]: data = adult.data
target = adult.target
target_names = adult.target_names
feature_names = adult.feature_names
category_map = adult.category_map
```

Note that for your own datasets you can use our utility function `gen_category_map` to create the category map.

```
[4]: from alibi.utils import gen_category_map

[5]: np.random.seed(0)
data_perm = np.random.permutation(np.c_[data, target])
data = data_perm[:, :-1]
target = data_perm[:, -1]
```



```
[6]: idx = 30000
X_train, y_train = data[:idx, :], target[:idx]
X_test, y_test = data[idx+1: :, :], target[idx+1:]
```

Create feature transformation pipeline

Create feature pre-processor. Needs to have ‘fit’ and ‘transform’ methods. Different types of pre-processing can be applied to all or part of the features. In the example below we will standardize ordinal features and apply one-hot-encoding to categorical features.

Ordinal features:

```
[7]: ordinal_features = [x for x in range(len(feature_names)) if x not in list(category_map.
    ↪keys())]
ordinal_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])
```

Categorical features:

```
[8]: categorical_features = list(category_map.keys())
categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
    ('onehot', OneHotEncoder(drop='first', handle_
    ↪unknown='error'))])
```

Note that in order to be able to interpret the coefficients corresponding to the categorical features, the option `drop='first'` has been passed to the `OneHotEncoder`. This means that for a categorical variable with `n` levels, the length of the code will be `n-1`. This is necessary in order to avoid introducing feature multicollinearity, which would skew the interpretation of the results. For more information about the issue about multicollinearity in the context of linear modelling see [\[1\]](#).

Combine and fit:

```
[9]: preprocessor = ColumnTransformer(transformers=[('num', ordinal_transformer, ordinal_
    ↪features),
    ('cat', categorical_transformer,
    ↪categorical_features)])
preprocessor.fit(X_train)
```

```
[9]: ColumnTransformer(transformers=[('num',
    Pipeline(steps=[('imputer',
        SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())]),
    [0, 8, 9, 10]),
    ('cat',
    Pipeline(steps=[('imputer',
        SimpleImputer(strategy='median')),
        ('onehot',
        OneHotEncoder(drop='first'))]),
    [1, 2, 3, 4, 5, 6, 7, 11])])
```

Fit a binary logistic regression classifier to the preprocessed Adult dataset

Preprocess the data

```
[10]: X_train_proc = preprocessor.transform(X_train)
      X_test_proc = preprocessor.transform(X_test)
```

Training

```
[11]: classifier = LogisticRegression(multi_class='multinomial',
                                     random_state=0,
                                     max_iter=500,
                                     verbose=0,
                                     )
      classifier.fit(X_train_proc, y_train)

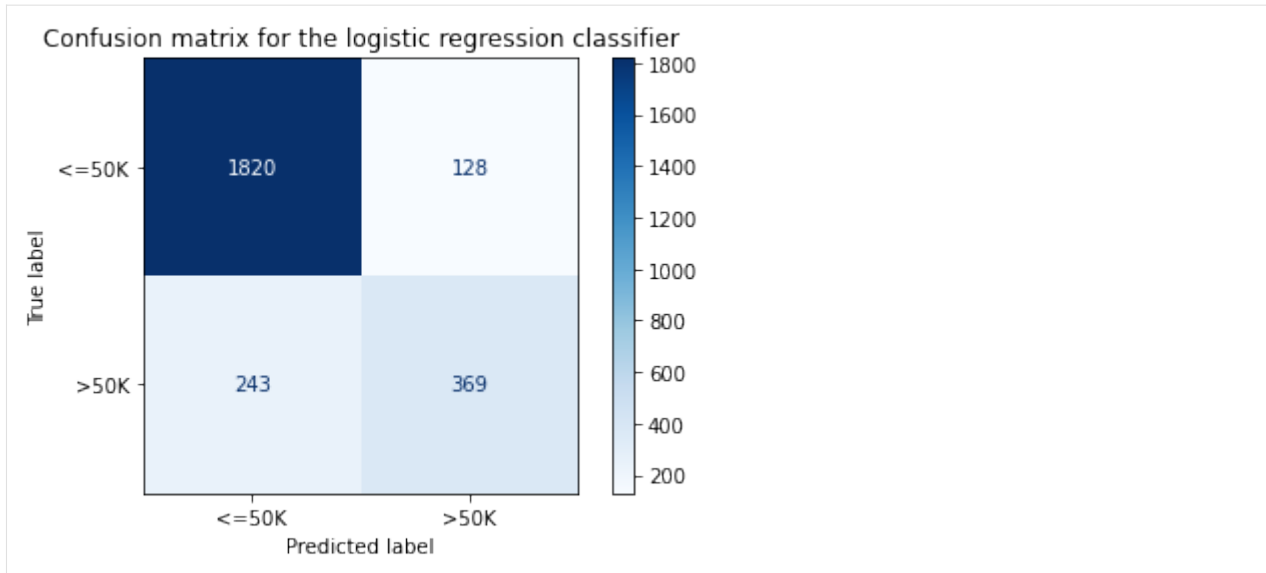
[11]: LogisticRegression(max_iter=500, multi_class='multinomial', random_state=0)
```

Model assessment

```
[12]: y_pred = classifier.predict(X_test_proc)

[13]: cm = confusion_matrix(y_test, y_pred)

[14]: title = 'Confusion matrix for the logistic regression classifier'
      disp = ConfusionMatrixDisplay.from_estimator(classifier,
                                                    X_test_proc,
                                                    y_test,
                                                    display_labels=target_names,
                                                    cmap=plt.cm.Blues,
                                                    normalize=None,
                                                    )
      disp.ax_.set_title(title);
```



```
[15]: print('Test accuracy: ', accuracy_score(y_test, classifier.predict(X_test_proc)))
```

```
Test accuracy:  0.855078125
```

Explaining the model with an explainer fitted on the preprocessed data

To speed up computation, we will use a background dataset with only 100 samples.

```
[16]: start_example_idx = 0
      stop_example_idx = 100
      background_data = slice(start_example_idx, stop_example_idx)
```

First, we group the categorical variables.

```
[17]: def make_groups(num_feats_names, cat_feats_names, feat_enc_dim):
      """
      Given a list with numerical feat. names, categorical feat. names
      and a list specifying the lengths of the encoding for each cat.
      variable, the function outputs a list of group names, and a list
      of the same len where each entry represents the column indices that
      the corresponding categorical feature
      """

      group_names = num_feats_names + cat_feats_names
      groups = []
      cat_var_idx = 0

      for name in group_names:
          if name in num_feats_names:
              groups.append(list(range(len(groups), len(groups) + 1)))
          else:
              start_idx = groups[-1][-1] + 1 if groups else 0
              groups.append(list(range(start_idx, start_idx + feat_enc_dim[cat_var_idx] )))
              cat_var_idx += 1
```

(continues on next page)

(continued from previous page)

```

    return group_names, groups

def sparse2ndarray(mat, examples=None):
    """
    Converts a scipy.sparse.csr.csr_matrix to a numpy.ndarray.
    If specified, examples is slice object specifying which selects a
    number of rows from mat and converts only the respective slice.
    """

    if examples:
        return mat[examples, :].toarray()

    return mat.toarray()

```

```

[18]: # obtain the indices of the categorical and the numerical features from the pipeline.
numerical_feats_idx = preprocessor.transformers_[0][2]
categorical_feats_idx = preprocessor.transformers_[1][2]
num_feats_names = [feature_names[i] for i in numerical_feats_idx]
cat_feats_names = [feature_names[i] for i in categorical_feats_idx]
perm_feat_names = num_feats_names + cat_feats_names
ohe = preprocessor.transformers_[1][1].named_steps['onehot']
feat_enc_dim = [len(cat_enc) - 1 for cat_enc in ohe.categories_]

```

```

[19]: # create the groups
X_train_proc_d = sparse2ndarray(X_train_proc, examples=background_data)
group_names, groups = make_groups(num_feats_names, cat_feats_names, feat_enc_dim)

```

Having created the groups, we are now ready to instantiate the explainer and explain our set.

```

[20]: pred_fcn = classifier.predict_proba
grp_lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names,
    ↪seed=0)
grp_lr_explainer.fit(X_train_proc_d, group_names=group_names, groups=groups)

[20]: KernelShap(meta={'name': 'KernelShap', 'type': 'blackbox', 'explanations': ['local',
    ↪'global'], 'params': {'groups': [[0], [1], [2], [3], [4, 5, 6, 7, 8, 9, 10, 11], [12,
    ↪13, 14, 15, 16, 17], [18, 19, 20], [21, 22, 23, 24, 25, 26, 27, 28], [29, 30, 31, 32,
    ↪33], [34, 35, 36, 37], [38], [39, 40, 41, 42, 43, 44, 45, 46, 47, 48]], 'group_names':
    ↪['Age', 'Capital Gain', 'Capital Loss', 'Hours per week', 'Workclass', 'Education',
    ↪'Marital Status', 'Occupation', 'Relationship', 'Race', 'Sex', 'Country'], 'weights':
    ↪None, 'summarise_background': False}})

```

We select only a small fraction of the testing set to explain for the purposes of this example.

```

[21]: def split_set(X, y, fraction, random_state=0):
    """
    Given a set X, associated labels y, split\|s a fraction y from X.
    """
    _, X_split, _, y_split = train_test_split(X,
                                                y,
                                                test_size=fraction,

```

(continues on next page)

(continued from previous page)

```

        random_state=random_state,
    )
    print("Number of records: {}".format(X_split.shape[0]))
    print("Number of class {}: {}".format(0, len(y_split) - y_split.sum()))
    print("Number of class {}: {}".format(1, y_split.sum()))

    return X_split, y_split

```

```

[22]: fraction_explained = 0.01
      X_explain, y_explain = split_set(X_test,
                                     y_test,
                                     fraction_explained,
                                     )
      X_explain_proc = preprocessor.transform(X_explain)
      X_explain_proc_d = sparse2ndarray(X_explain_proc)

      Number of records: 26
      Number of class 0: 20
      Number of class 1: 6

```

```
[ ]: grouped_explanation = grp_lr_explainer.explain(X_explain_proc_d)
```

Explaining with an explainer fitted on the raw data

To explain with an explainer fitted on the raw data, we make the preprocessor part of the predictor, as shown below.

```

[24]: pred_fcn = lambda x: classifier.predict_proba(preprocessor.transform(x))
      lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=feature_names, seed=0)

```

We use the same background dataset to fit the explainer.

```

[25]: lr_explainer.fit(X_train[background_data])

[25]: KernelShap(meta={'name': 'KernelShap', 'type': 'blackbox', 'explanations': ['local',
→ 'global'], 'params': {'groups': None, 'group_names': None, 'weights': None, 'summarise_
→ background': False}})

```

We explain the same dataset as before.

```
[ ]: explanation = lr_explainer.explain(X_explain)
```

Results comparison

To show that fitting the explainer on the raw data and combining the preprocessor with the classifier gives the same results as grouping the variables and fitting the explainer on the preprocessed data, we check to see that the same features are considered as most important when combining the two approaches.

```

[27]: def get_ranked_values(explanation):
      """
      Retrieves a tuple of (feature_effects, feature_names) for

```

(continues on next page)

(continued from previous page)

```

each class explained. A feature's effect is its average
shap value magnitude across an array of instances.
"""

ranked_shap_vals = []
for cls_idx in range(len(explanation.shap_values)):
    this_ranking = (
        explanation.raw['importances'][str(cls_idx)][['ranked_effect']],
        explanation.raw['importances'][str(cls_idx)][['names']]
    )
    ranked_shap_vals.append(this_ranking)

return ranked_shap_vals

def compare_ranking(ranking_1, ranking_2, methods=None):
    for i, (combined, grouped) in enumerate(zip(ranking_1, ranking_2)):
        print(f"Class: {i}")
        c_names, g_names = combined[1], grouped[1]
        c_mag, g_mag = combined[0], grouped[0]
        different = []
        for i, (c_n, g_n) in enumerate(zip(c_names, g_names)):
            if c_n != g_n:
                different.append((i, c_n, g_n))
        if different:
            method_1 = methods[0] if methods else "Method_1"
            method_2 = methods[1] if methods else "Method_2"
            i, c_ns, g_ns = list(zip(*different))
            data = {"Rank": i, method_1: c_ns, method_2: g_ns}
            df = pd.DataFrame(data=data)
            print("Found the following rank differences:")
            print(df)
        else:
            print("The methods provided the same ranking for the feature effects.")
            print(f"The ranking is: {c_names}")
        print("")

def reorder_feats(vals_and_names, src_vals_and_names):
    """Given a two tuples, each containing a list of ranked feature
    shap values and the corresponding feature names, the function
    reorders the values in vals according to the order specified in
    the list of names contained in src_vals_and_names.
    """

    _, src_names = src_vals_and_names
    vals, names = vals_and_names
    reordered = np.zeros_like(vals)

    for i, name in enumerate(src_names):
        alt_idx = names.index(name)
        reordered[i] = vals[alt_idx]

    return reordered, src_names

```

(continues on next page)

(continued from previous page)

```

def compare_avg_mag_shap(class_idx, comparisons, baseline, **kwargs):
    """
    Given a list of tuples, baseline, containing the feature values and a list with
    ↪ feature names
    for each class and, comparisons, a list of lists with tuples with the same structure
    ↪, the
    function reorders the values of the features in comparisons entries according to the
    ↪ order
    of the feature names provided in the baseline entries and displays the feature
    ↪ values for comparison.
    """

    methods = kwargs.get("methods", [f"method_{i}" for i in range(len(comparisons) + 1)])

    n_features = len(baseline[class_idx][0])

    # bar settings
    bar_width = kwargs.get("bar_width", 0.05)
    bar_space = kwargs.get("bar_space", 2)

    # x axis
    x_low = kwargs.get("x_low", 0.0)
    x_high = kwargs.get("x_high", 1.0)
    x_step = kwargs.get("x_step", 0.05)
    x_ticks = np.round(np.arange(x_low, x_high + x_step, x_step), 3)

    # y axis (these are the y coordinate of start and end of each group
    # of bars)
    start_y_pos = np.array(np.arange(0, n_features))*bar_space
    end_y_pos = start_y_pos + bar_width*len(methods)
    y_ticks = 0.5*(start_y_pos + end_y_pos)

    # figure
    fig_x = kwargs.get("fig_x", 10)
    fig_y = kwargs.get("fig_y", 7)

    # fontsizes
    title_font = kwargs.get("title_fontsize", 20)
    legend_font = kwargs.get("legend_fontsize", 20)
    tick_labels_font = kwargs.get("tick_labels_fontsize", 20)
    axes_label_fontsize = kwargs.get("axes_label_fontsize", 10)

    # labels
    title = kwargs.get("title", None)
    ylabel = kwargs.get("ylabel", None)
    xlabel = kwargs.get("xlabel", None)

    # process input data
    methods = list(reversed(methods))
    base_vals = baseline[class_idx][0]
    ordering = baseline[class_idx][1]

```

(continues on next page)

(continued from previous page)

```

comp_vals = []

# reorder the features so that they match the order of the baseline (ordering)
for comparison in comparisons:
    vals, ord_ = reorder_feats(comparison[class_idx], baseline[class_idx])
    comp_vals.append(vals)
    assert ord_ is ordering

all_vals = [base_vals] + comp_vals
data = dict(zip(methods, all_vals))
df = pd.DataFrame(data=data, index=ordering)

# plotting logic
fig, ax = plt.subplots(figsize=(fig_x, fig_y))

for i, col in enumerate(df.columns):
    values = list(df[col])
    y_pos = [y + bar_width*i for y in start_y_pos]
    ax.barh(y_pos, list(values), bar_width, label=col)

# add ticks, legend and labels
ax.set_xticks(x_ticks)
ax.set_xticklabels([str(x) for x in x_ticks], rotation=45, fontsize=tick_labels_font)
ax.set_xlabel(xlabel, fontsize=axes_label_fontsize)
ax.set_yticks(y_ticks)
ax.set_yticklabels(ordering, fontsize=tick_labels_font)
ax.set_ylabel(ylabel, fontsize=axes_label_fontsize)
ax.invert_yaxis() # labels read top-to-bottom
ax.legend(fontsize=legend_font)

plt.grid(True)
plt.title(title, fontsize=title_font)

return ax, fig, df

```

```

[28]: ranked_grouped_shap_vals = get_ranked_values(grouped_explanation)
ranked_shal_vals_raw = get_ranked_values(explanation)
compare_ranking(ranked_grouped_shap_vals, ranked_shal_vals_raw)

```

```

Class: 0
The methods provided the same ranking for the feature effects.
The ranking is: ['Marital Status', 'Capital Gain', 'Education', 'Occupation', 'Sex',
↳ 'Relationship', 'Age', 'Hours per week', 'Workclass', 'Capital Loss', 'Country', 'Race
↳ ']

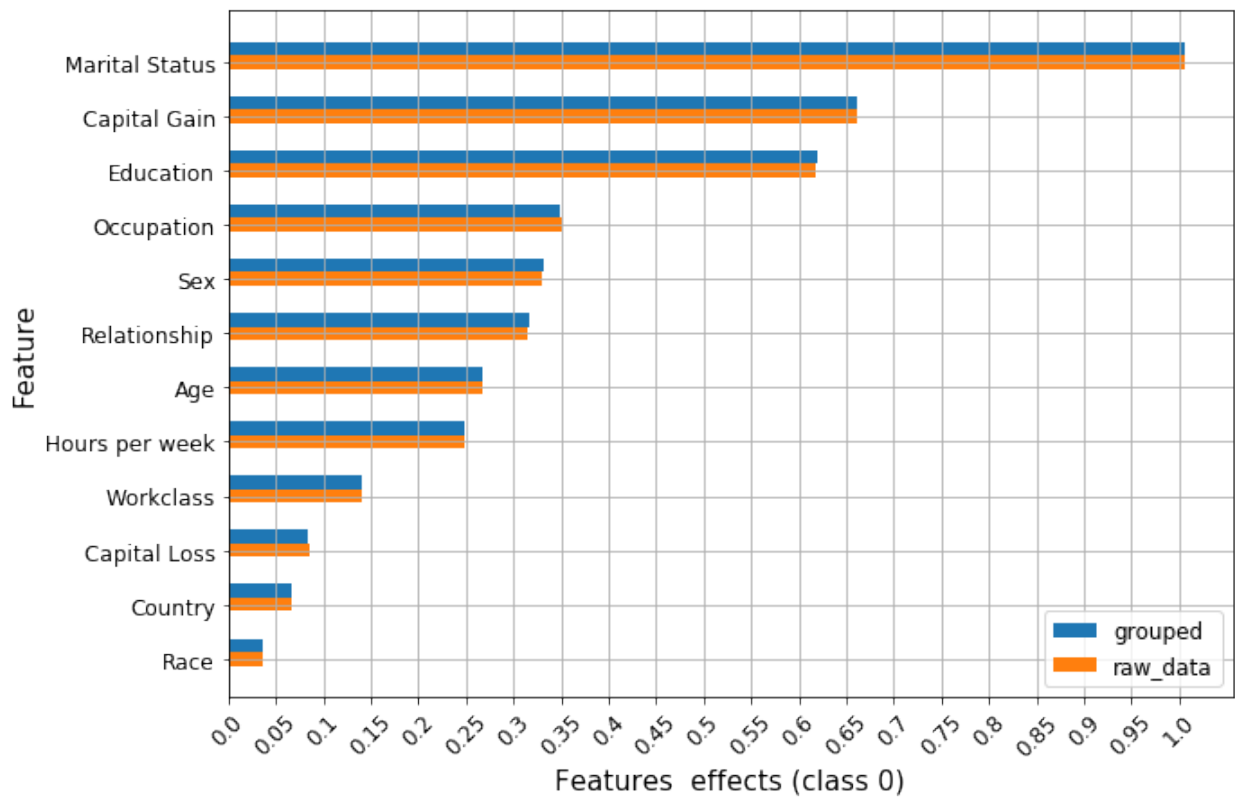
Class: 1
The methods provided the same ranking for the feature effects.
The ranking is: ['Marital Status', 'Capital Gain', 'Education', 'Occupation', 'Sex',
↳ 'Relationship', 'Age', 'Hours per week', 'Workclass', 'Capital Loss', 'Country', 'Race
↳ ']

```

Above we can see that both methods returned the same feature importances.


```
[29]: class_idx = 0

ax, fig, _ = compare_avg_mag_shap(class_idx,
                                   [ranked_shap_vals_raw],
                                   ranked_grouped_shap_vals,
                                   methods=('raw_data', 'grouped'),
                                   bar_width=0.5,
                                   tick_labels_fontsize=12,
                                   legend_fontsize=12,
                                   title_fontsize=15,
                                   xlabel="Features effects (class {})".format(0),
                                   ylabel="Feature",
                                   axes_label_fontsize=15,
                                   )
```



We can see that the shap values are very similar. The differences appear because the regression dataset generated in order to compute the shap values differs slightly between the two runs due to the difference in the order of the features in the background dataset.

References

[1] Mahto, K.K., 2019. “One-Hot-Encoding, Multicollinearity and the Dummy Variable Trap”. Retrieved 02 Feb 2020 ([link](#))

8.9.3 Handling categorical variables with KernelSHAP

Note

To enable SHAP support, you may need to run

```
pip install alibi[shap]
```

```
[ ]: # shap.summary_plot currently doesn't work with matplotlib>=3.6.0,  
# see bug report: https://github.com/slundberg/shap/issues/2687  
!pip install matplotlib==3.5.3
```

Introduction

In this example, we show how the KernelSHAP method can be used for tabular data, which contains both numerical (continuous) and categorical attributes. Using a logistic regression model fitted to the `Adult` dataset, we examine the performance of the KernelSHAP algorithm against the exact shap values. We investigate the effect of the background dataset size on the estimated shap values and present two ways of handling categorical data.

```
[ ]: import shap  
shap.initjs()  
  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd  
  
from alibi.explainers import KernelShap  
from alibi.datasets import fetch_adult  
from scipy.special import logit  
from sklearn.compose import ColumnTransformer  
from sklearn.impute import SimpleImputer  
from sklearn.linear_model import LogisticRegression  
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay  
from sklearn.model_selection import cross_val_score, train_test_split  
from sklearn.pipeline import Pipeline  
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

Data preparation

Load and split

The `fetch_adult` function returns a Bunch object containing the features, the targets, the feature names and a mapping of categorical variables to numbers.

```
[2]: adult = fetch_adult()
      adult.keys()

[2]: dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])
```

```
[3]: data = adult.data
      target = adult.target
      target_names = adult.target_names
      feature_names = adult.feature_names
      category_map = adult.category_map
```

Note that for your own datasets you can use our utility function `gen_category_map` to create the category map.

```
[4]: from alibi.utils import gen_category_map

[5]: np.random.seed(0)
      data_perm = np.random.permutation(np.c_[data, target])
      data = data_perm[:, :-1]
      target = data_perm[:, -1]

[6]: idx = 30000
      X_train, y_train = data[:idx, :], target[:idx]
      X_test, y_test = data[idx+1:, :], target[idx+1:]
```

Create feature transformation pipeline

Create feature transformation pipeline

Create feature pre-processor. Needs to have ‘fit’ and ‘transform’ methods. Different types of pre-processing can be applied to all or part of the features. In the example below we will standardize ordinal features and apply one-hot-encoding to categorical features.

Ordinal features:

```
[7]: ordinal_features = [x for x in range(len(feature_names)) if x not in list(category_map.
    ↪ keys())]
      ordinal_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
    ↪ ('scaler', StandardScaler())])
```

Categorical features:

```
[8]: categorical_features = list(category_map.keys())
      categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
    ↪ ('onehot', OneHotEncoder(drop='first', handle_
    ↪ unknown='error'))])
```

Note that in order to be able to interpret the coefficients corresponding to the categorical features, the option `drop='first'` has been passed to the `OneHotEncoder`. This means that for a categorical variable with `n` levels, the length of the code will be `n-1`. This is necessary in order to avoid introducing feature multicollinearity, which would skew the interpretation of the results. For more information about the issue about multicollinearity in the context of linear modelling see [\[1\]](#).

Combine and fit:

```
[9]: preprocessor = ColumnTransformer(transformers=[('num', ordinal_transformer, ordinal_
    ↪ features),
                                           ('cat', categorical_transformer, ↪
    ↪ categorical_features)])
preprocessor.fit(X_train)
```

```
[9]: ColumnTransformer(transformers=[('num',
    Pipeline(steps=[('imputer',
        SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())]),
    [0, 8, 9, 10]),
    ('cat',
    Pipeline(steps=[('imputer',
        SimpleImputer(strategy='median')),
        ('onehot',
        OneHotEncoder(drop='first'))]),
    [1, 2, 3, 4, 5, 6, 7, 11])])
```

Preprocess the data

```
[10]: X_train_proc = preprocessor.transform(X_train)
X_test_proc = preprocessor.transform(X_test)
```

Fit a binary logistic regression classifier to the Adult dataset

Training

```
[11]: classifier = LogisticRegression(multi_class='multinomial',
    random_state=0,
    max_iter=500,
    verbose=0,
    )
classifier.fit(X_train_proc, y_train)
```

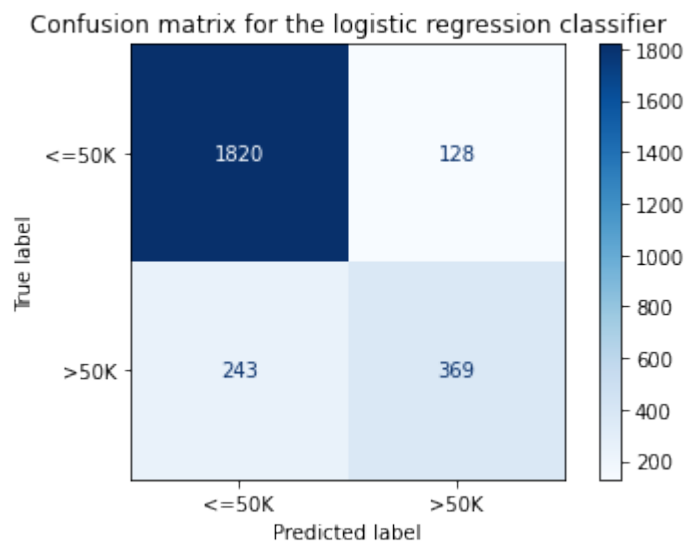
```
[11]: LogisticRegression(max_iter=500, multi_class='multinomial', random_state=0)
```

Model assessment

```
[12]: y_pred = classifier.predict(X_test_proc)
```

```
[13]: cm = confusion_matrix(y_test, y_pred)
```

```
[14]: title = 'Confusion matrix for the logistic regression classifier'
disp = ConfusionMatrixDisplay.from_estimator(classifier,
                                             X_test_proc,
                                             y_test,
                                             display_labels=target_names,
                                             cmap=plt.cm.Blues,
                                             normalize=None,
                                             )
disp.ax_.set_title(title);
```



```
[15]: print('Test accuracy: ', accuracy_score(y_test, classifier.predict(X_test_proc)))
```

```
Test accuracy: 0.855078125
```

Interpreting the logistic regression model

In order to interpret the logistic regression model, we need to first recover the encoded feature names. The feature effect of a categorical variable is computed by summing the coefficients of the encoded variables. Hence, we first understand how the preprocessing transformation acts on the data and then obtain the overall effects from the model coefficients.

First, we are concerned with understanding the dimensionality of a preprocessed record and what it is comprised of.

```
[16]: idx = 0
print(f"The dimensionality of a preprocessed record is {X_train_proc[idx:idx+1, :].shape}")
print(f"Then number of continuous features in the original data is {len(ordinal_features)}")
```

The dimensionality of a preprocessed record is (1, 49).
Then number of continuos features in the original data is 4.

Therefore, of 49, 45 of the dimensions of the original data are encoded categorical features. We obtain `feat_enc_dim`, an array with the lengths of the encoded dimensions for each categorical variable that will be use for processing the results later on.

```
[17]: fts = [feature_names[x] for x in categorical_features]
      # get feature names for the encoded categorical features
      ohe = preprocessor.transformers_[1][1].named_steps['onehot']
      cat_enc_feat_names = ohe.get_feature_names_out(fts)
      # compute encoded dimension; -1 as ohe is setup with drop='first'
      feat_enc_dim = [len(cat_enc) - 1 for cat_enc in ohe.categories_]
      d = {'feature_names': fts, 'encoded_dim': feat_enc_dim}
      df = pd.DataFrame(data=d)
      print(df)
      total_dim = df['encoded_dim'].sum()
      print(f"The dimensionality of the encoded categorical features is {total_dim}.")
      assert total_dim == len(cat_enc_feat_names)
```

	feature_names	encoded_dim
0	Workclass	8
1	Education	6
2	Marital Status	3
3	Occupation	8
4	Relationship	5
5	Race	4
6	Sex	1
7	Country	10

The dimensionality of the encoded categorical features is 45.

By analysing an encoded record, we can recover the mapping of column indices to the features they represent.

```
[18]: print(X_train_proc[0, :])

(0, 0)      -0.8464456331823879
(0, 1)      -0.14513571926899238
(0, 2)      -0.21784551572515998
(0, 3)       0.28898151525672766
(0, 7)       1.0
(0, 15)      1.0
(0, 19)      1.0
(0, 21)      1.0
(0, 32)      1.0
(0, 37)      1.0
(0, 47)      1.0
```

```
[19]: numerical_feats_idx = preprocessor.transformers_[0][2]
      categorical_feats_idx = preprocessor.transformers_[1][2]
      scaler = preprocessor.transformers_[0][1].named_steps['scaler']
      print((X_train[idx, numerical_feats_idx] - scaler.mean_)/scaler.scale_)
      num_feats_names = [feature_names[i] for i in numerical_feats_idx]
      cat_feats_names = [feature_names[i] for i in categorical_feats_idx]
      print(num_feats_names)
```

```
[-0.84644563 -0.14513572 -0.21784552  0.28898152]
['Age', 'Capital Gain', 'Capital Loss', 'Hours per week']
```

Therefore, the first four columns of the encoded data represent the Age, Capital Gain Capital Loss and Hours per week features. Notice these features have a different index in the dataset prior to processing `X_train`.

The remainder of the columns encode the encoded categorical features, as shown below.

```
[20]: print(cat_enc_feat_names)

['Workclass_1.0' 'Workclass_2.0' 'Workclass_3.0' 'Workclass_4.0'
'Workclass_5.0' 'Workclass_6.0' 'Workclass_7.0' 'Workclass_8.0'
'Education_1.0' 'Education_2.0' 'Education_3.0' 'Education_4.0'
'Education_5.0' 'Education_6.0' 'Marital Status_1.0' 'Marital Status_2.0'
'Marital Status_3.0' 'Occupation_1.0' 'Occupation_2.0' 'Occupation_3.0'
'Occupation_4.0' 'Occupation_5.0' 'Occupation_6.0' 'Occupation_7.0'
'Occupation_8.0' 'Relationship_1.0' 'Relationship_2.0' 'Relationship_3.0'
'Relationship_4.0' 'Relationship_5.0' 'Race_1.0' 'Race_2.0' 'Race_3.0'
'Race_4.0' 'Sex_1.0' 'Country_1.0' 'Country_2.0' 'Country_3.0'
'Country_4.0' 'Country_5.0' 'Country_6.0' 'Country_7.0' 'Country_8.0'
'Country_9.0' 'Country_10.0']
```

To obtain a single coefficient for each categorical variable, we pass a list with the indices where each encoded categorical variable starts and the encodings dimensions to the `sum_categories` function.

```
[21]: from alibi.explainers.shap_wrappers import sum_categories
```

Compute the start index of each categorical variable knowing that the categorical variables are adjacent and follow the continuous features.

```
[22]: start=len(ordinal_features)
cat_feat_start = [start]
for dim in feat_enc_dim[:-1]:
    cat_feat_start.append(dim + cat_feat_start[-1])
```

```
[23]: beta = classifier.coef_
beta = np.concatenate((-beta, beta), axis=0)
intercepts = classifier.intercept_
intercepts = np.concatenate((-intercepts, intercepts), axis=0)
all_coef = sum_categories(beta, cat_feat_start, feat_enc_dim)
```

Extract and plot feature importances. Please see [this](#) example for background on interpreting logistic regression coefficients.

```
[24]: def get_importance(class_idx, beta, feature_names, intercepts=None):
    """
    Retrive and sort abs magnitude of coefficients from model.
    """

    # sort the absolute value of model coef from largest to smallest
    srt_beta_k = np.argsort(np.abs(beta[class_idx, :]))[::-1]
    feat_names = [feature_names[idx] for idx in srt_beta_k]
    feat_imp = beta[class_idx, srt_beta_k]
    # include bias among feat importances
```

(continues on next page)

(continued from previous page)

```

    if intercepts is not None:
        intercept = intercepts[class_idx]
        bias_idx = len(feats_imp) - np.searchsorted(np.abs(feats_imp)[::-1], np.
↪abs(intercept) )
        feats_imp = np.insert(feats_imp, bias_idx, intercept.item(), )
        intercept_idx = np.where(feats_imp == intercept)[0][0]
        feat_names.insert(intercept_idx, 'bias')

    return feats_imp, feat_names

def plot_importance(feats_imp, feat_names, class_idx, **kwargs):
    """
    Create a horizontal barchart of feature effects, sorted by their magnitude.
    """

    left_x, right_x = kwargs.get("left_x"), kwargs.get("right_x")
    eps_factor = kwargs.get("eps_factor", 4.5)

    fig, ax = plt.subplots(figsize=(10, 5))
    y_pos = np.arange(len(feats_imp))
    ax.barh(y_pos, feats_imp)
    ax.set_yticks(y_pos)
    ax.set_yticklabels(feat_names, fontsize=15)
    ax.invert_yaxis() # labels read top-to-bottom
    ax.set_xlabel(f'Feature effects for class {class_idx}', fontsize=15)
    ax.set_xlim(left=left_x, right=right_x)

    for i, v in enumerate(feats_imp):
        eps = 0.03
        if v < 0:
            eps = -eps_factor*eps
        ax.text(v + eps, i + .25, str(round(v, 3)))

    return ax, fig

```

```

[25]: class_idx = 0
      perm_feat_names = num_feats_names + cat_feats_names

```

```

[26]: perm_feat_names # feats are reordered by preprocessor

```

```

[26]: ['Age',
      'Capital Gain',
      'Capital Loss',
      'Hours per week',
      'Workclass',
      'Education',
      'Marital Status',
      'Occupation',
      'Relationship',
      'Race',
      'Sex',
      'Country']

```

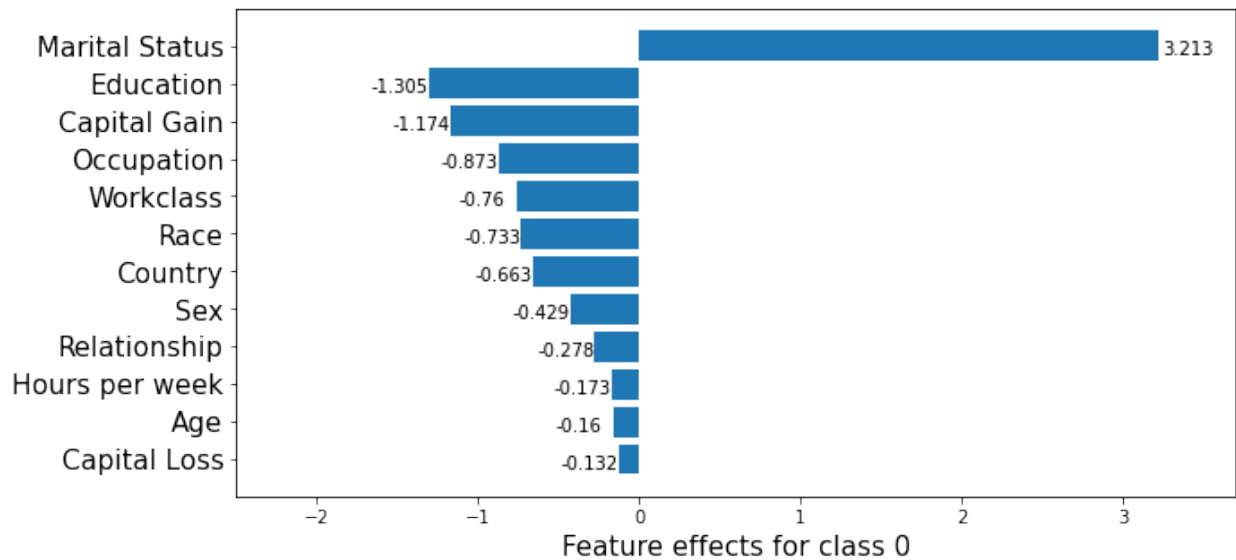


```
[27]: feat_imp, srt_feat_names = get_importance(class_idx,
                                             all_coef,
                                             perm_feat_names,
                                             )
```

```
[28]: srt_feat_names
```

```
[28]: ['Marital Status',
      'Education',
      'Capital Gain',
      'Occupation',
      'Workclass',
      'Race',
      'Country',
      'Sex',
      'Relationship',
      'Hours per week',
      'Age',
      'Capital Loss']
```

```
[29]: _, class_0_fig = plot_importance(feat_imp,
                                       srt_feat_names,
                                       class_idx,
                                       left_x=-2.5,
                                       right_x=3.7,
                                       eps_factor=12 # controls text distance from end of bar
                                       )
```



Note that in the above, the feature effects are with respect to the model bias, which has a value of 1.31.

```
[30]: # Sanity check to ensure graph is correct.
print(beta[class_idx, 0:4]) # Age, Capital Gains, Capital Loss, Hours per week
print(np.sum(beta[class_idx, 18:21])) # Marital status
```

```
[-0.15990831 -1.17397349 -0.13215877 -0.17288254]
3.2134915799542094
```

Apply KernelSHAP to explain the model

Note that the *local accuracy* property of SHAP (eq. (5) in [1]) requires

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i. \quad (1)$$

Hence, sum of the feature importances should be equal to the model output, $f(x)$. By passing `link='logit'` to the explainer, we ensure that ϕ_0 , the *base value* (see **Local explanation** section [here](#)) will be calculated in the margin space (i.e., a logit transformation is applied to the probabilities) where the logistic regression model is additive.

Further considerations when applying the KernelSHAP method to this dataset are:

- **the background dataset size:** by setting a larger value for the `stop_example_idx` in the set below, you can observe how the runtime of the algorithm increases. At the same time, it is important to have a diverse but sufficiently large set of samples as background so that the missing feature values are correctly integrated. A way to reduce the number of samples is to pass the `summarise_background=True` flag to the explainer `fit` option along with the desired number of samples (`n_background_samples`). If there are no categorical variables in the data and there is no data grouping, then a k-means clustering algorithm is used to summarise the data. Otherwise, the data is sampled uniformly at random. Below, we used the `train_test_split` function of `sklearn` instead so that the label proportions are approximately the same as in the original split.
- **the number of instances to be explained:** the test set contains a number of 2560 records, which are 49-dimensional after pre-processing, as opposed to 13-dimensional as in the Wine dataset example. For this reason, only a fraction of `fraction_explained` (default 5%) are explained by way of getting a more general view of the model behaviour compared to simply analysing local explanations
- **treating the encoded categorical features as a group** of features that are **jointly** perturbed as opposed to being perturbed individually

```
[31]: def split_set(X, y, fraction, random_state=0):
    """
    Given a set X, associated labels y, splits a fraction y from X.
    """
    _, X_split, _, y_split = train_test_split(X,
                                              y,
                                              test_size=fraction,
                                              random_state=random_state,
                                              )
    print(f"Number of records: {X_split.shape[0]}")
    print(f"Number of class {0}: {len(y_split) - y_split.sum()}")
    print(f"Number of class {1}: {y_split.sum()}")

    return X_split, y_split
```

```
[32]: fraction_explained = 0.05
X_explain, y_explain = split_set(X_test,
                                y_test,
                                fraction_explained,
                                )
X_explain_proc = preprocessor.transform(X_explain)
```

```

Number of records: 128
Number of class 0: 96
Number of class 1: 32

```

```

[33]: # Select only 100 examples for the background dataset to speedup computation
start_example_idx = 0
stop_example_idx = 100
background_data = slice(start_example_idx, stop_example_idx)

```

Exploiting explanation model additivity to estimate the effects of categorical features

Inspired by equation (1), a way to estimate the overall effect of a categorical variable is to treat its encoded levels as individual binary variables and sum the estimated effects for the encoded dimensions.

```

[34]: pred_fcn = classifier.predict_proba
lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names)
lr_explainer.fit(X_train_proc[background_data, :])

```

```

[34]: KernelShap(meta={
    'name': 'KernelShap',
    'type': ['blackbox'],
    'explanations': ['local', 'global'],
    'params': {
        'groups': None,
        'group_names': None,
        'weights': None,
        'summarise_background': False
    }
})

```

```

[35]: # passing the logit link function to the explainer ensures the units are consistent ...
mean_scores_train = logit(pred_fcn(X_train_proc[background_data, :]).mean(axis=0))
# print(mean_scores_train - lr_explainer.expected_value)

```

```

[36]: lr_explainer.expected_value

```

```

[36]: array([ 1.08786649, -1.08786649])

```

```

[ ]: explanation = lr_explainer.explain(X_explain_proc,
                                     summarise_result=True,
                                     cat_vars_start_idx=cat_feat_start,
                                     cat_vars_enc_dim=feat_enc_dim,
                                     )

```

We now sum the estimate shap values for each dimension to obtain one shap value for each categorical variable!

```

[39]: def rank_features(shap_values, feat_names):
    """
    Given an NxF array of shap values where N is the number of
    instances and F number of features, the function ranks the
    shap values according to their average magnitude.
    """

```

(continues on next page)

(continued from previous page)

```

"""

avg_mag = np.mean(np.abs(shap_values), axis=0)
srt = np.argsort(avg_mag)[::-1]
rank_values = avg_mag[srt]
rank_names = [feat_names[idx] for idx in srt]

return rank_values, rank_names

def get_ranked_values(explanation):
    """
    Retrives a tuple of (feature_effects, feature_names) for
    each class explained. A feature's effect is its average
    shap value magnitude across an array of instances.
    """

    ranked_shap_vals = []
    for cls_idx in range(len(explanation.shap_values)):
        this_ranking = (
            explanation.raw['importances'][str(cls_idx)]['ranked_effect'],
            explanation.raw['importances'][str(cls_idx)]['names']
        )
        ranked_shap_vals.append(this_ranking)

    return ranked_shap_vals

```

```
[40]: ranked_combined_shap_vals = get_ranked_values(explanation)
```

Because the columns have been permuted by the preprocessor, the columns of the instances to be explained have to be permuted before creating the summary plot.

```
[41]: perm_feat_names
```

```
[41]: ['Age',
'Capital Gain',
'Capital Loss',
'Hours per week',
'Workclass',
'Education',
'Marital Status',
'Occupation',
'Relationship',
'Race',
'Sex',
'Country']
```

```
[42]: def permute_columns(X, feat_names, perm_feat_names):
    """
    Permutes the original dataset so that its columns
    (ordered according to feat_names) have the order

```

(continues on next page)

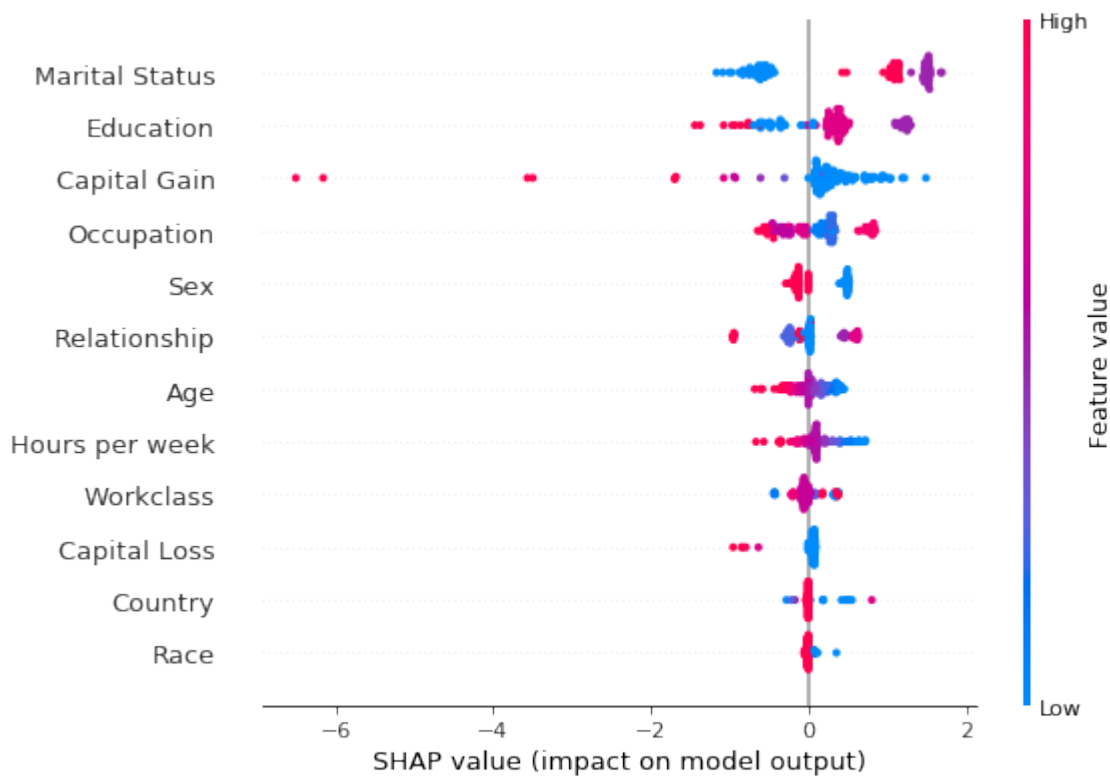
(continued from previous page)

```
of the variables after transformation with the
sklearn preprocessing pipeline (perm_feat_names).
"""
```

```
perm_X = np.zeros_like(X)
perm = []
for i, feat_name in enumerate(perm_feat_names):
    feat_idx = feat_names.index(feat_name)
    perm_X[:, i] = X[:, feat_idx]
    perm.append(feat_idx)
return perm_X, perm
```

```
[43]: perm_X_explain, _ = permute_columns(X_explain, feature_names, perm_feat_names)
```

```
[44]: shap.summary_plot(explanation.shap_values[0], perm_X_explain, perm_feat_names)
```



Note that the aggregated local explanations of this limited set are in partial agreement with the global explanation provided by the model coefficients. The top 3 most important features are determined to be the same. We can see that, high values of the Capital Gains decrease the odds of a sample being classified as `class_0` (income <\$50k).

Grouping features with KernelShap

An alternative way to deal with one-hot encoded categorical variables is to group the levels of a categorical variables and treat them as a single variable during the sampling process that generates the training data for the explanation model. Dealing with the categorical variables in this way can help reduce the variance of the shap values estimate (1). Note that this does not *necessarily* result in a runtime saving: by default the algorithm estimates the shap values by creating a training dataset for the weighed regression, which consists of tiling `nsamples` (2) copies of the background dataset. By default, this parameter is set to `auto`, which is given by $2 * M + 2 * 11$ where M is the number of features which can be perturbed. Therefore, because $2 * M < 2 * 11$, one should not expect to see significant time savings when reducing the number of columns. The runtime can be improved by reducing `nsamples` at the cost of a loss in estimation accuracy. (3)

The following arguments should be passed to the `fit` step in order to perform grouping:

- `background_data`: in this case, `X_train_proc`(4)
- `group_names`: a list containing the feature names
- `groups`: for each feature name in `group_name`, `groups` contains a list of column indices in `X_train_proc` which represent that feature.

```
[45]: def make_groups(num_feats_names, cat_feats_names, feat_enc_dim):
    """
    Given a list with numerical feat. names, categorical feat. names
    and a list specifying the lengths of the encoding for each cat.
    variable, the function outputs a list of group names, and a list
    of the same len where each entry represents the column indices that
    the corresponding categorical feature
    """

    group_names = num_feats_names + cat_feats_names
    groups = []
    cat_var_idx = 0

    for name in group_names:
        if name in num_feats_names:
            groups.append(list(range(len(groups), len(groups) + 1)))
        else:
            start_idx = groups[-1][-1] + 1 if groups else 0
            groups.append(list(range(start_idx, start_idx + feat_enc_dim[cat_var_idx] )))
            cat_var_idx += 1

    return group_names, groups

def sparse2ndarray(mat, examples=None):
    """
    Converts a scipy.sparse.csr.csr_matrix to a numpy.ndarray.
    If specified, examples is slice object specifying which selects a
    number of rows from mat and converts only the respective slice.
    """

    if examples:
        return mat[examples, :].toarray()

    return mat.toarray()
```

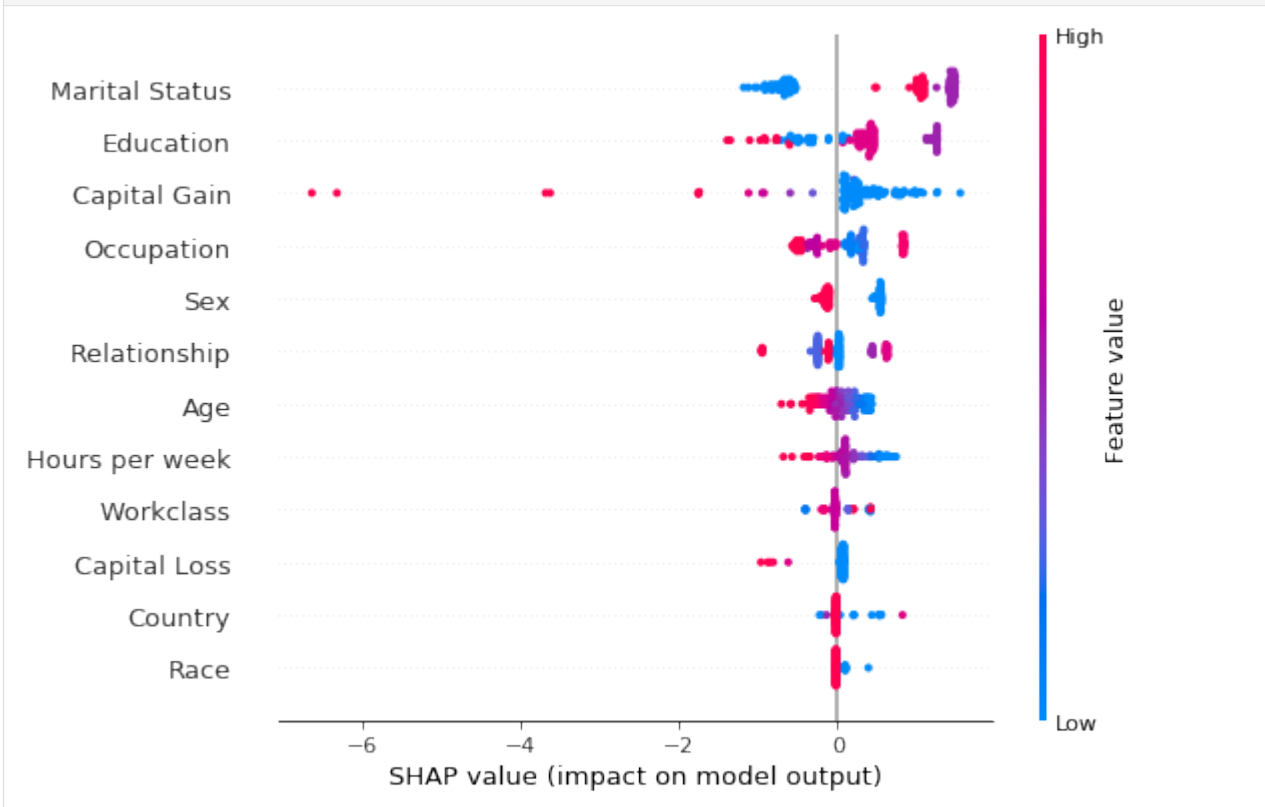
```
[46]: X_train_proc_d = sparse2ndarray(X_train_proc, examples=background_data)
group_names, groups = make_groups(num_feats_names, cat_feats_names, feat_enc_dim)
```

Having created the groups, we are now ready to instantiate the explainer and explain our set.

```
[48]: X_explain_proc_d = sparse2ndarray(X_explain_proc)
grp_lr_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names)
grp_lr_explainer.fit(X_train_proc_d, group_names=group_names, groups=groups)
```

```
[ ]: grouped_explanation = grp_lr_explainer.explain(X_explain_proc_d)
```

```
[50]: shap.summary_plot(grouped_explanation.shap_values[0], perm_X_explain, perm_feat_names)
```



```
[51]: ranked_grouped_shap_vals = get_ranked_values(grouped_explanation)
```

Having ranked the features by the average magnitude of their shap value, we can now see if they provide the same ranking. Yet another way to deal with the categorical variables is to fit the explainer to the unprocessed dataset and combine the preprocessor with the predictor. We show this approach yields the same results in [this](#) example.

```
[52]: def compare_ranking(ranking_1, ranking_2, methods=None):
    for i, (combined, grouped) in enumerate(zip(ranking_1, ranking_2)):
        print(f"Class: {i}")
        c_names, g_names = combined[1], grouped[1]
        c_mag, g_mag = combined[0], grouped[0]
        different = []
        for i, (c_n, g_n) in enumerate(zip(c_names, g_names)):
            if c_n != g_n:
```

(continues on next page)

(continued from previous page)

```

        different.append((i, c_n, g_n))
    if different:
        method_1 = methods[0] if methods else "Method_1"
        method_2 = methods[1] if methods else "Method_2"
        i, c_ns, g_ns = list(zip(*different))
        data = {"Rank": i, method_1: c_ns, method_2: g_ns}
        df = pd.DataFrame(data=data)
        print("Found the following rank differences:")
        print(df)
    else:
        print("The methods provided the same ranking for the feature effects.")
        print(f"The ranking is: {c_names}")
    print("")

compare_ranking(ranked_combined_shap_vals, ranked_grouped_shap_vals)

Class: 0
The methods provided the same ranking for the feature effects.
The ranking is: ['Marital Status', 'Education', 'Capital Gain', 'Occupation', 'Sex',
↳ 'Relationship', 'Age', 'Hours per week', 'Workclass', 'Capital Loss', 'Country', 'Race
↳ ']

Class: 1
The methods provided the same ranking for the feature effects.
The ranking is: ['Marital Status', 'Education', 'Capital Gain', 'Occupation', 'Sex',
↳ 'Relationship', 'Age', 'Hours per week', 'Workclass', 'Capital Loss', 'Country', 'Race
↳ ']

```

As shown in [this](#) example, for a logistic regression model, the exact shap values can be computed as shown below. Note that, like KernelShap, this computation makes the assumption that the features are independent.

```

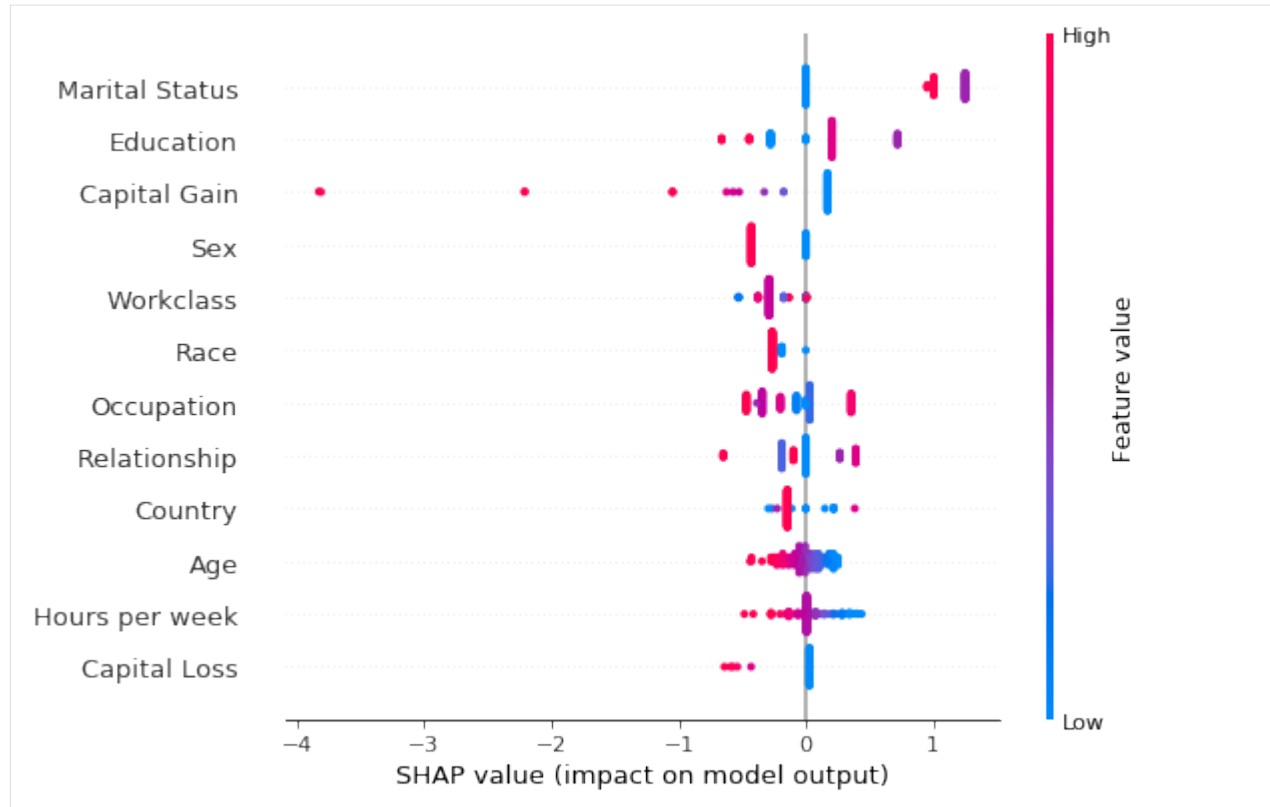
[67]: exact_shap = [(beta[:, None, :]*X_explain_proc_d)[i, ...] for i in range(beta.shape[0])]
combined_exact_shap = [sum_categories(shap_values, cat_feat_start, feat_enc_dim) for
↳ shap_values in exact_shap]
ranked_combined_exact_shap = [rank_features(vals, perm_feat_names) for vals in combined_
↳ exact_shap]

```

```

[58]: shap.summary_plot(combined_exact_shap[0], perm_X_explain, perm_feat_names )

```

Comparing the two summary plots above, we notice that albeit the estimation and the exact method rank the features `Marital Status`, `Education` and `Capital Gain` as the features that are most important for the classification decision, the ranking of the remainder of the features differs. In particular, while `Race` is estimated to be the sixth more important feature using the exact shap value computation, it is deemed as the least important in the approximate computation. However, note that the exact shap value calculation takes into account the weight estimated by the logistic regression model. All the weights in the model are estimated jointly so that the model predictive distribution matches the predictive distribution of the training data. Thus, the values of the coefficients are a function of the entire dataset. On the other hand, to limit the computation time, the shap values are estimated using a small background dataset. This error is compounded by the fact that the estimation is approximate, since computing the exact values using the weighted regression has exponential computational complexity. Below, we show that the `Race` feature distribution is heavily skewed towards white individuals. Investigating correcting this imbalance would lead to more accurate estimation is left to future work.

```
[69]: from functools import partial
      from collections import Counter
```

```
[70]: def get_feature_distribution(dataset, feature, category_map, feature_names):
      """Given a map of categorical variable indices to human-readable
      values and an array of feature integer values, the function outputs
      the distribution the feature in human readable format."""
      feat_mapping = category_map[feature_names.index(feature)]
      distrib_raw = Counter(dataset)
      distrib = {feat_mapping[key]: val for key, val in distrib_raw.items()}

      return distrib
```

```
[71]: get_distribution = partial(get_feature_distribution, feature_names=feature_names,
    ↪ category_map=category_map)
    race_idx = feature_names.index("Race")
    bkg_race_distrib = get_distribution(X_train[background_data, race_idx], 'Race')
    train_race_distrib = get_distribution(X_train[:, race_idx], 'Race')
    expl_race_distrib = get_distribution(X_explain[:, race_idx], 'Race')
```

```
[72]: print("Background data race distribution:")
    print(bkg_race_distrib)
    print("Train data race distribution:")
    print(train_race_distrib)
    print("Explain race distribution:")
    print(expl_race_distrib)
```

```
Background data race distribution:
{'White': 89, 'Amer-Indian-Eskimo': 2, 'Black': 8, 'Asian-Pac-Islander': 1}
Train data race distribution:
{'White': 25634, 'Amer-Indian-Eskimo': 285, 'Black': 2868, 'Asian-Pac-Islander': 963,
 ↪ 'Other': 250}
Explain race distribution:
{'White': 105, 'Black': 20, 'Asian-Pac-Islander': 2, 'Amer-Indian-Eskimo': 1}
```

We now look to compare the approximate and the exact shap values as well as the relation between the shap computation and the logistic regression coefficients.

```
[372]: def reorder_feats(vals_and_names, src_vals_and_names):
    """Given a two tuples, each containing a list of ranked feature
    shap values and the corresponding feature names, the function
    reorders the values in vals according to the order specified in
    the list of names contained in src_vals_and_names.
    """

    _, src_names = src_vals_and_names
    vals, names = vals_and_names
    reordered = np.zeros_like(vals)

    for i, name in enumerate(src_names):
        alt_idx = names.index(name)
        reordered[i] = vals[alt_idx]

    return reordered, src_names

def compare_avg_mag_shap(class_idx, comparisons, baseline, **kwargs):
    """
    Given a list of tuples, baseline, containing the feature values and a list with
    ↪ feature names
    for each class and, comparisons, a list of lists with tuples with the same structure
    ↪, the
    function reorders the values of the features in comparisons entries according to the
    ↪ order
    of the feature names provided in the baseline entries and displays the feature
    ↪ values for comparison.
    """
```

(continues on next page)

(continued from previous page)

```

methods = kwargs.get("methods", [f"method_{i}" for i in range(len(comparisons) + 1)])

n_features = len(baseline[class_idx][0])

# bar settings
bar_width = kwargs.get("bar_width", 0.05)
bar_space = kwargs.get("bar_space", 2)

# x axis
x_low = kwargs.get("x_low", 0.0)
x_high = kwargs.get("x_high", 1.0)
x_step = kwargs.get("x_step", 0.05)
x_ticks = np.round(np.arange(x_low, x_high + x_step, x_step), 3)

# y axis (these are the y coordinate of start and end of each group
# of bars)
start_y_pos = np.array(np.arange(0, n_features))*bar_space
end_y_pos = start_y_pos + bar_width*len(methods)
y_ticks = 0.5*(start_y_pos + end_y_pos)

# figure
fig_x = kwargs.get("fig_x", 10)
fig_y = kwargs.get("fig_y", 7)

# fontsizes
title_font = kwargs.get("title_fontsize", 20)
legend_font = kwargs.get("legend_fontsize", 20)
tick_labels_font = kwargs.get("tick_labels_fontsize", 20)
axes_label_fontsize = kwargs.get("axes_label_fontsize", 10)

# labels
title = kwargs.get("title", None)
ylabel = kwargs.get("ylabel", None)
xlabel = kwargs.get("xlabel", None)

# process input data
methods = list(reversed(methods))
base_vals = baseline[class_idx][0]
ordering = baseline[class_idx][1]
comp_vals = []

# reorder the features so that they match the order of the baseline (ordering)
for comparison in comparisons:
    vals, ord_ = reorder_feats(comparison[class_idx], baseline[class_idx])
    comp_vals.append(vals)
    assert ord_ is ordering

all_vals = [base_vals] + comp_vals
data = dict(zip(methods, all_vals))
df = pd.DataFrame(data=data, index=ordering)

```

(continues on next page)

(continued from previous page)

```

# plotting logic
fig, ax = plt.subplots(figsize=(fig_x, fig_y))

for i, col in enumerate(df.columns):
    values = list(df[col])
    y_pos = [y + bar_width*i for y in start_y_pos]
    ax.barh(y_pos, list(values), bar_width, label=col)

# add ticks, legend and labels
ax.set_xticks(x_ticks)
ax.set_xticklabels([str(x) for x in x_ticks], rotation=45, fontsize=tick_labels_font)
ax.set_xlabel(xlabel, fontsize=axes_label_fontsize)
ax.set_yticks(y_ticks)
ax.set_yticklabels(ordering, fontsize=tick_labels_font)
ax.set_ylabel(ylabel, fontsize=axes_label_fontsize)
ax.invert_yaxis() # labels read top-to-bottom
ax.legend(fontsize=legend_font)

plt.grid(True)
plt.title(title, fontsize=title_font)

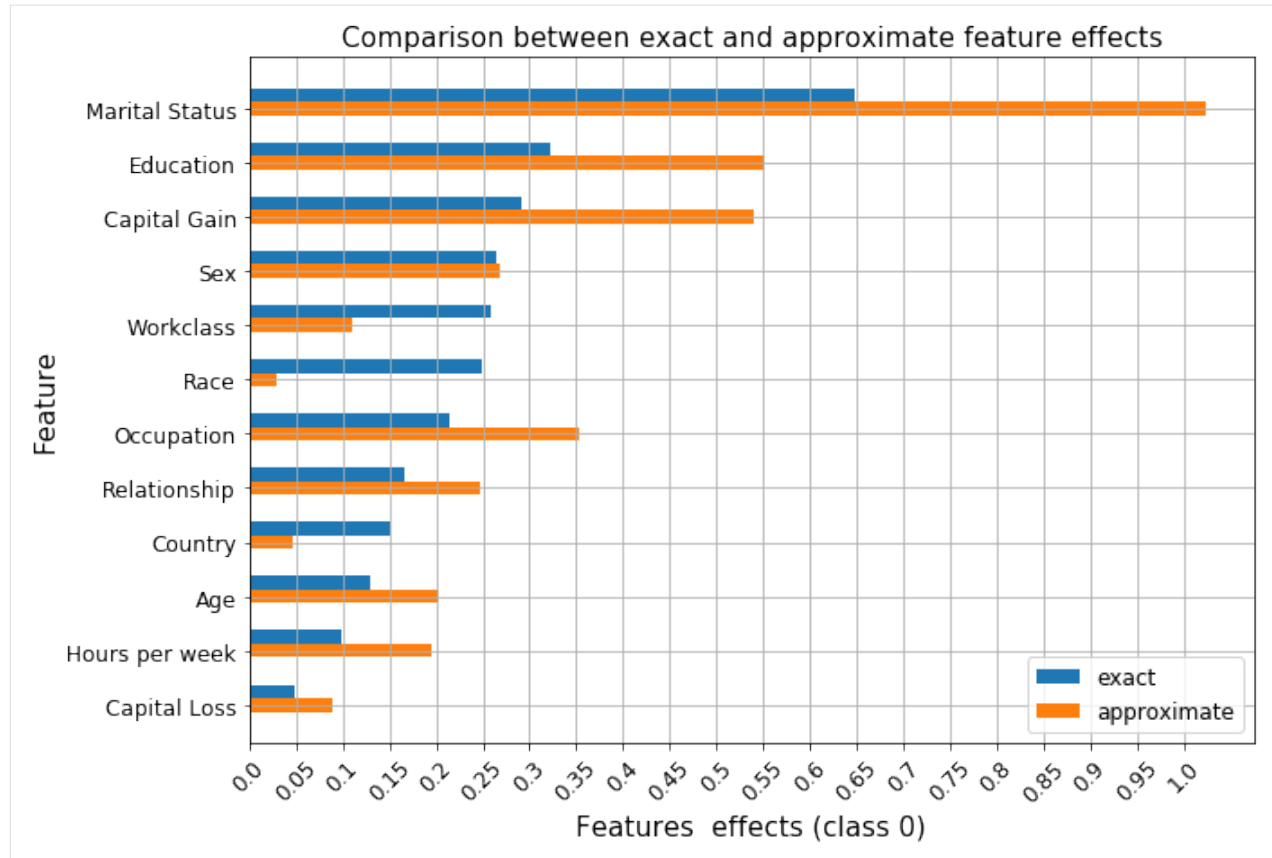
return ax, fig, df

```

```

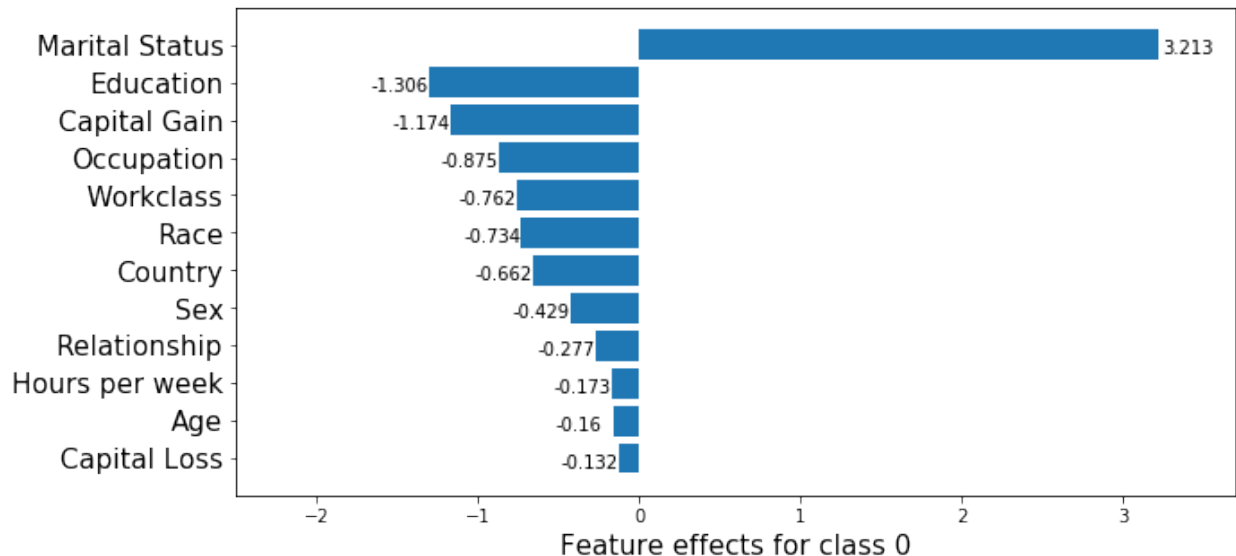
[356]: class_idx = 0
ax, fig, _ = compare_avg_mag_shap(class_idx,
                                  [ranked_combined_shap_vals],
                                  ranked_combined_exact_shap,
                                  methods=('approximate', 'exact'),
                                  bar_width=0.5,
                                  tick_labels_fontsize=12,
                                  legend_fontsize=12,
                                  title="Comparison between exact and approximate feature_
→effects",
                                  title_fontsize=15,
                                  xlabel=f"Features effects (class {0})",
                                  ylabel="Feature",
                                  axes_label_fontsize=15,
                                  )

```



```
[75]: class_0_fig
```

```
[75]:
```



As before, we see that features such as `Occupation`, `Workclass` or `Race` have similar effects according to the ranking of the logistic regression coefficients and that the exact shap value estimation recovers this effect since it is computed using the underlying coefficients. Unlike in our previous example, these relationships are not recovered by the approximate estimation procedure. Therefore, whenever possible, exact shap value computation should be preferred to approximations. As shown in this example it is possible to calculate exact shap values for linear models and exact algo-

rithms exist for tree models. The approximate procedure still gives insights into the model, but, as shown above, it can be quite sensitive when the effects of the variables are similar. The notable differences between the two explanations are the importance of the `Race` and `Country` are underestimated by a significant margin and their rank significantly differs from the exact computation.

Finally, as noted in [4] as the model bias (7) increases, more weight can be assigned to irrelevant features. This is perhaps expected since a linear model will suffer from bias when applied to data generated from a nonlinear process, so we don't expect the feature effects to be accurately estimated. This also affects the exact shap values, which depend on these coefficients.

Investigating the feature effects given a range of feature values

Given an individual record, one could ask questions of the type *What would have been the effect of feature x had its value been y ?*. To answer this question one can create hypothetical instances starting from a base record, where the hypothetical instances have a different value for a chosen feature than the original record. Below, we study the effect of the `Capital Gain` feature as a function of its value. We choose the 0th record in the `X_explain` set, which represents an individual with no capital gain.

```
[76]: idx = 0
base_record = X_explain[idx, ]
cap_gain = X_explain[idx, feature_names.index('Capital Gain')]
print(f"The capital gain of individual {idx} is {cap_gain}")
```

```
The capital gain of individual 0 is 0!
```

We now create a dataset of records that differ from a base record only by the `Capital Gain` feature.

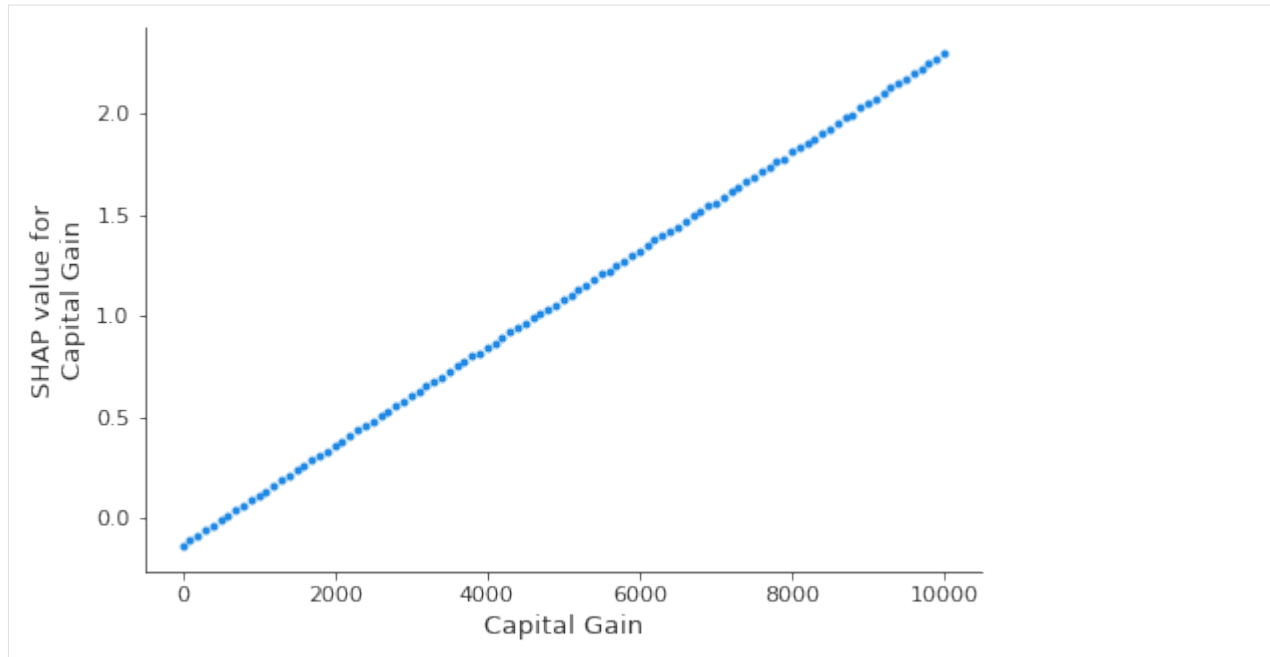
```
[77]: cap_increment = 100
cap_range = range(0, 10100, cap_increment)
hyp_record = np.repeat(base_record[None, :], len(cap_range), axis=0)
hyp_record[:, feature_names.index('Capital Gain')] = cap_range
assert (hyp_record[1, :] - hyp_record[0, :]).sum() == cap_increment
X_hyp_proc = preprocessor.transform(hyp_record)
X_hyp_proc_d = X_hyp_proc.toarray()
```

We can explain the hypothetical instances in order to understand the change in the `Capital Gain` effect as a function of its value.

```
[ ]: hyp_explainer = KernelShap(pred_fcn, link='logit', feature_names=perm_feat_names)
hyp_explainer.fit(X_train_proc_d, group_names=group_names, groups=groups)
hyp_explanation = hyp_explainer.explain(X_hyp_proc_d)
```

```
[79]: hyp_record_perm, _ = permute_columns(hyp_record, feature_names, perm_feat_names)
```

```
[80]: shap.dependence_plot('Capital Gain',
                           hyp_explanation.shap_values[1],
                           hyp_record_perm,
                           feature_names=perm_feat_names,
                           interaction_index=None,
                           )
```



In a logistic regression model, the predictors are linearly related to the logits. Estimating the shap values using the KernelShap clearly recovers this aspect, as shown by the plot above. The dependence of the feature effect on the feature value has important implications on the shap value estimation; since the model relies on using the background dataset to simulate the effect of *missing* inputs in order to estimate any feature effect, it is important to select an appropriate background dataset in order to avoid biasing the estimate of the feature effect of interest. Below, we will experiment with the size of the background dataset, split from the training set of the classifier while keeping the class representation proportions of the training set roughly the same.

An alternative way to display the effect of a value as a function of the feature value is to group the similar prediction paths, which can be done by specifying the `hclust` feature ordering option.

```
[81]: # obtain the human readable version of the base record (for display purposes)
base_perm, perm = permute_columns(base_record[None, :], feature_names, perm_feat_names)
br = []
for i, x in enumerate(np.nditer(base_record.squeeze())):
    if i in categorical_features:
        br.append(category_map[i][x])
    else:
        br.append(x.item())
br = [br[i] for i in perm]
df = pd.DataFrame(data=np.array(br).reshape(1, -1), columns=perm_feat_names)
```

```
[82]: df
```

```
[82]:   Age Capital Gain Capital Loss Hours per week Workclass  Education \
0   49             0             0             55   Private Prof-School

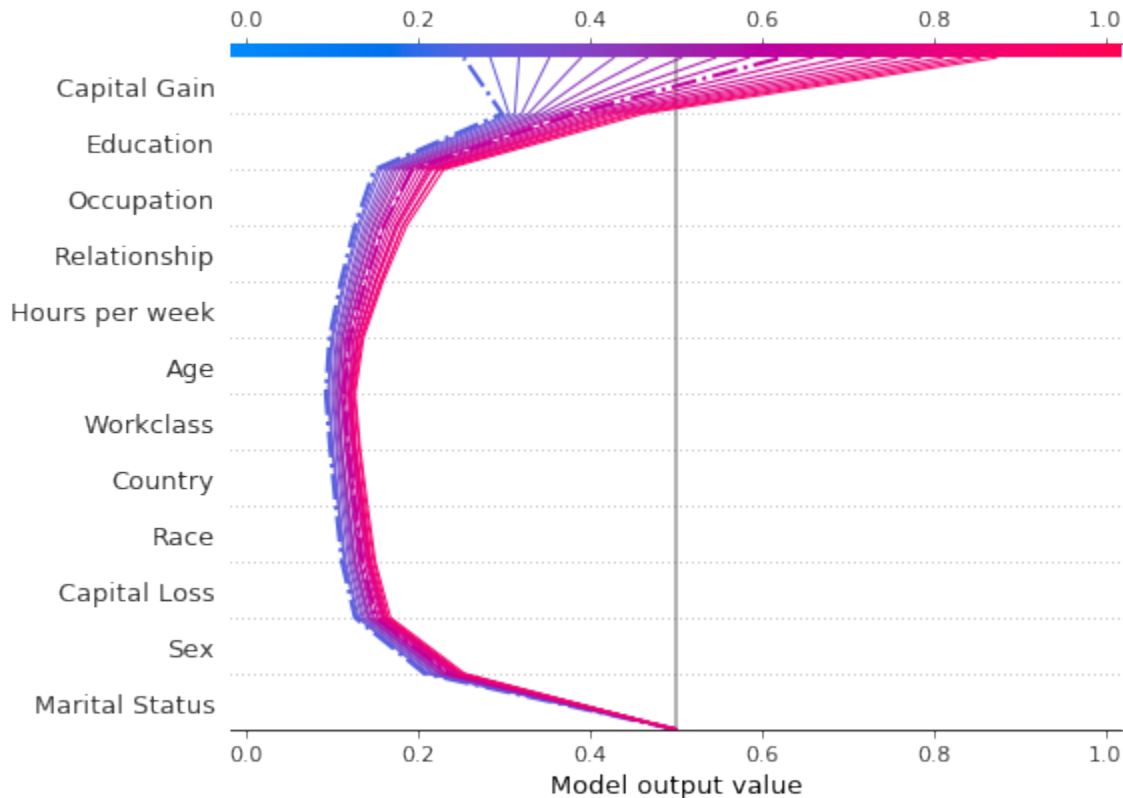
   Marital Status  Occupation  Relationship  Race  Sex  Country
0  Never-Married  Professional  Not-in-family  White  Female  United-States
```

```
[84]: r = shap.decision_plot(hyp_explainer.expected_value[1],
                           hyp_explanation.shap_values[1][0:-1:5],
```

(continues on next page)

(continued from previous page)

```
hyp_record_perm,
link='logit',
feature_names=perm_feat_names,
feature_order='hclust',
highlight=[0, 10],
new_base_value = 0.0,
return_objects=True)
```



```
[85]: hyp_record[0:-1:5][10,:]
```

```
[85]: array([[ 49,   4,   6,   1,   5,   1,   4,   0, 5000,   0,  55,
           9]])
```

The decision plot above informs us of the path to the decision `Income < $50,000` for the original record (depicted in blue, and, for clarity, on its own below). Additionally, decision paths for fictitious records where only the `Capital Gain` feature was altered are displayed. For clarity, only a handful of these instances have been plotted. Note that the base value of the plot has been altered to be the classification threshold (6) as opposed to the expected prediction probability for individuals earning more than \$50,000.

We see that the second highlighted instance (in purple) would have been predicted as making an income over \$50,000 with approximately 0.6 probability, and that this change in prediction is largely driven by the `Capital Gain` feature. We can see below that the income predictor would have predicted the income of this individual to be more than \$50,000 had the `Capital Gain` been over \$3,500.

```
[86]: # the 7th record from the filtered ones would be predicted to make an income > $50k
income_pred_proba = pred_fcn(preprocessor.transform(hyp_record[0:-1:5][7,:][None,:]))
print(f"Prediction probabilities: {income_pred_proba}")
```

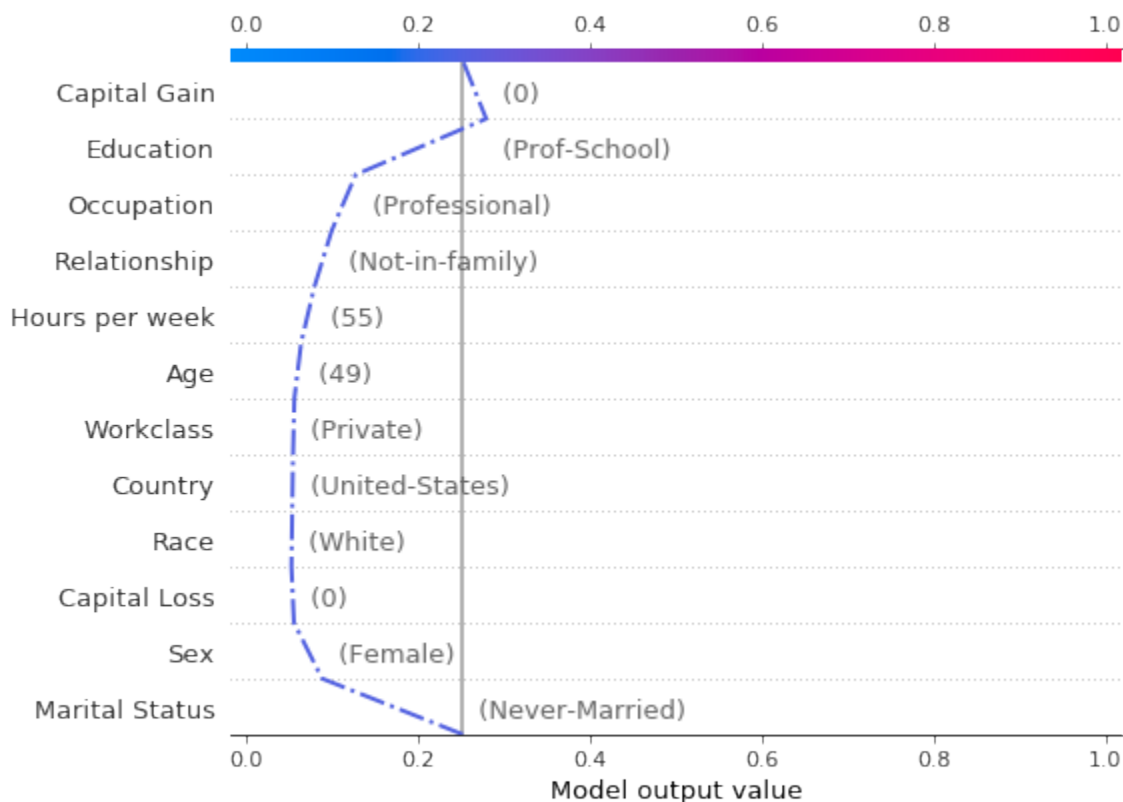
(continues on next page)

(continued from previous page)

```
# we can see that the minimum capital gain for the prediction to change is: $3,500
cap_gain_min = hyp_record[0:-1:5][7,feature_names.index('Capital Gain')]
print(f"Minimum capital gain is: ${cap_gain_min}")
```

```
Prediction probabilities: [[0.49346669 0.50653331]]
Minimum capital gain is: $3500
```

```
[88]: shap.decision_plot(hyp_explainer.expected_value[1],
                        hyp_explanation.shap_values[1][0],
                        df,
                        link='logit',
                        feature_order=r.feature_idx,
                        highlight=0
                        )
```



Note that passing `return_objects=True` and using the `r.feature_idx` as an input to the decision plot above we were able to plot the original record along with the feature values in the same feature order. Additionally, by passing `logit` to the plotting function, the scale of the axis is mapped from the margin to probability space(5).

Combined, the two decision plots show that:

- the largest decrease in the probability of earning more than \$50,000 is significantly affected if the individual is has marital status `Never-Married`
- the largest increase in the probability of earning more than \$50,000 is determined by the education level
- the probability of making an income greater than \$50,000 increases with the capital gain; notice how this implies that features such as `Education` or `Occupation` also contribute more to the increase in probability of earning more than \$50,000

Checking if prediction paths significantly differ for extreme probability predictions

One can employ the decision plot to check if the prediction paths for low (or high) probability examples differ significantly; conceptually, examples which exhibit prediction paths which are significantly different are potential outliers.

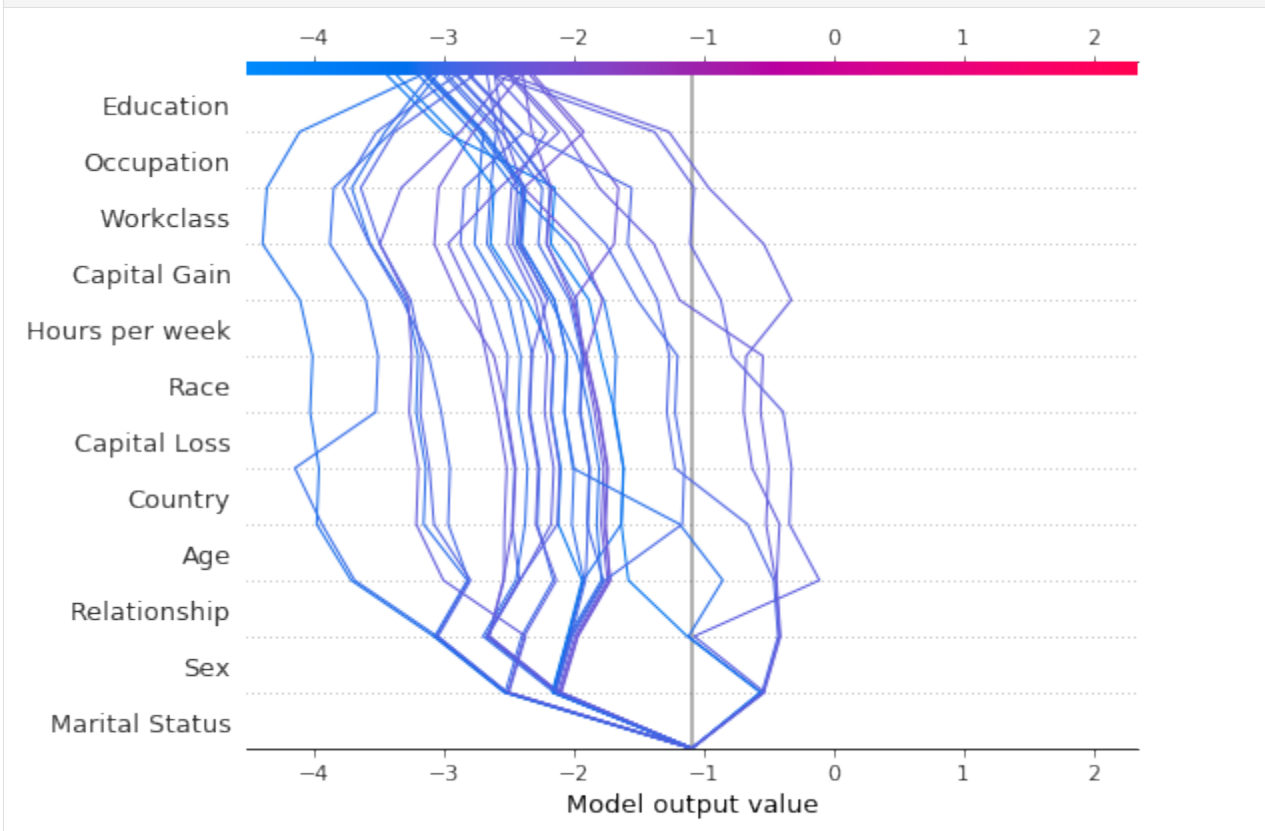
Below, we seek to explain only those examples which are predicted to have an income above \$ 50,000 with small probability.

```
[89]: predictions = classifier.predict_proba(X_explain_proc)
low_prob_idx = np.logical_and(predictions[:, 1] <= 0.1, predictions[:, 1] >= 0.03)
X_low_prob = X_explain_proc[low_prob_idx, :]
```

```
[ ]: low_prob_explanation = hyp_explainer.explain(X_low_prob.toarray())
```

```
[92]: X_low_prob_perm, _ = permute_columns(X_explain[low_prob_idx, :], feature_names, perm_
      ↪ feat_names)
```

```
shap.decision_plot(hyp_explainer.expected_value[1],
                  low_prob_explanation.shap_values[1],
                  X_low_prob_perm,
                  feature_names=perm_feat_names,
                  feature_order='hclust')
```



From the above plot, we see that the prediction paths for the samples with low probability of being class 1 are similar - no potential outliers are identified.

Investigating the effect of the background dataset size on shap value estimates

The shap values estimation relies on querying the model with samples where certain inputs are toggled off in order to infer the contribution of a particular feature. Since most models cannot accept arbitrary patterns of missing values, the background dataset is used to replace the values of the missing features, that is, as a *background model*. In more detail, the algorithm first creates a number of copies of this dataset, and then subsamples sets of

Since the model predicts on these perturbed samples and regresses on the predictions to infer the shap values, the quality of the background model is key for the explanation model. Here we will not be concerned with modelling the background, but instead investigate whether simply increasing the background set size can give rise to wildly different shap values. This part of the example is **long running** so the graph showing our original results can be loaded instead.

```
[93]: import pickle
```

```
[94]: def get_dataset(X_train, y_train, split_fraction):
    """
    Splits and transforms a dataset
    """

    split_X, _ = split_set(X_train, y_train, split_fraction)
    split_X_proc = preprocessor.transform(split_X)
    split_X_proc_d = sparse2ndarray(split_X_proc)

    return split_X_proc_d
```

Below cell is long running, skip and display the graph instead.

```
[ ]: split_fractions = [0.005, 0.01, 0.02, 0.04, 0.08, 0.16]
exp_data = {'data': [],
            'explainers': [],
            'raw_shap': [],
            'split_fraction': [],
            'ranked_shap_vals': [],
            }
fname = 'experiment.pkl'

for fraction in split_fractions:
    data = get_dataset(X_train, y_train, fraction)
    explainer = KernelShap(pred_fcn, link='logit')
    explainer.fit(data, group_names=group_names, groups=groups)
    explanation = explainer.explain(X_explain_proc_d)
    ranked_avg_shap = get_ranked_values(explanation)
    exp_data['data'].append(data)
    exp_data['explainers'].append(explainer)
    exp_data['raw_shap'].append(explanation.shap_values)
    exp_data['ranked_shap_vals'].append(ranked_avg_shap)
    with open(fname, 'wb') as f:
        pickle.dump(exp_data, f)
```

```
[ ]: comparisons = exp_data['ranked_shap_vals']
methods = [f'train_fraction={fr}' for fr in split_fractions] + ['exact']
_, fg, df = compare_avg_mag_shap(class_idx,
                                comparisons,
```

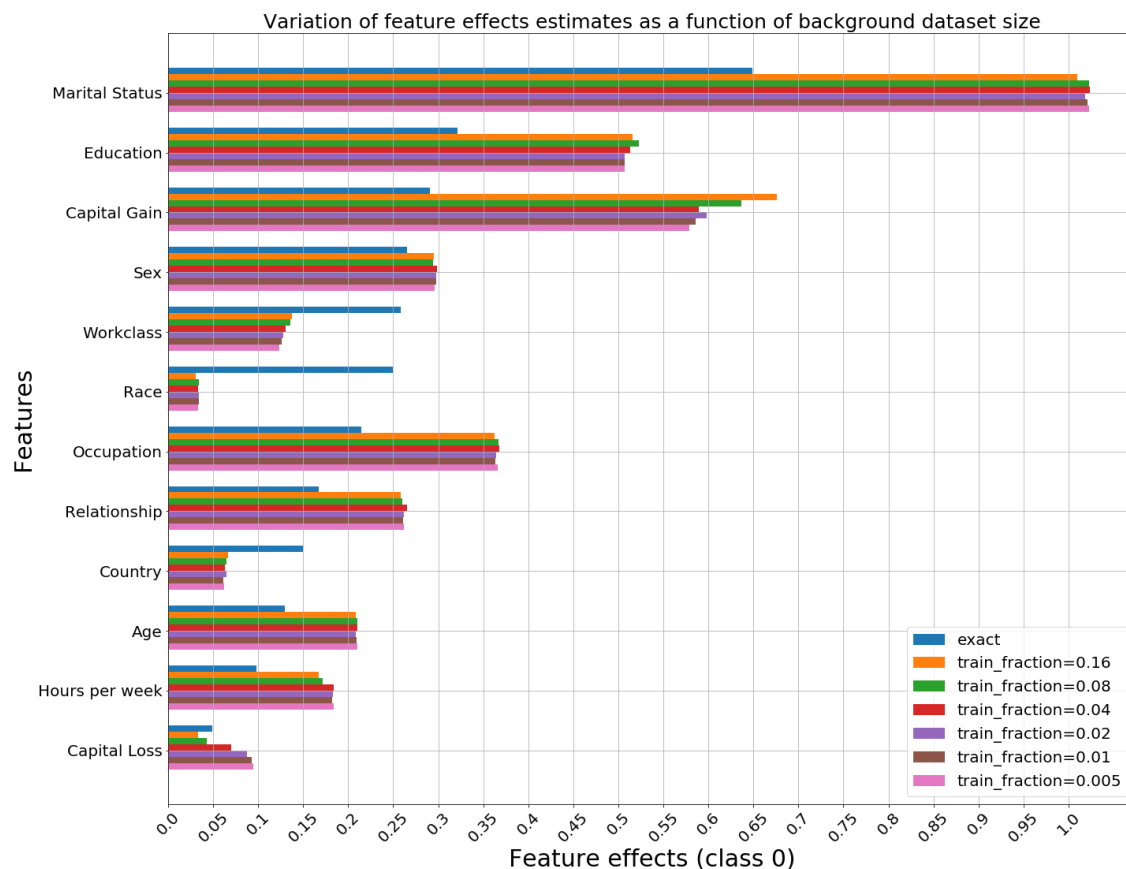
(continues on next page)

(continued from previous page)

```

ranked_combined_exact_shap,
methods=methods,
fig_x=22,
fig_y=18,
bar_width=1,
bar_space=9.5,
xlabel=f"Feature effects (class {0})",
ylabel="Features",
axes_label_fontsize=30,
title="Variation of feature effects estimates as a
↪function of background dataset size",
title_fontsize=30,
legend_fontsize=25,
)

```



We notice that with the exception of the Capital Gain and Capital Loss, the differences between the shap values estimates are not significant as the fraction of the training set used as a background dataset increases from 0.005 to 0.16. Notably, the Capital Gain feature would be ranked as the second most important by the all approximate models, whereas in the initial experiment which used the first 100 (0.003) examples from the training set the ranking of the two features was reversed. How to select an appropriate background dataset is an open ended question. In the future,

we will explore whether clustering the training data can provide a more representative background model and increase the accuracy of the estimation.

A potential limitation of expensive explanation methods such as KernelShap when used to draw insights about the global model behaviour is the fact that explaining large datasets can take a long time. Below, we explain a larger fraction of the testing set (0.4) in order to see if different conclusions about the feature importances would be made.

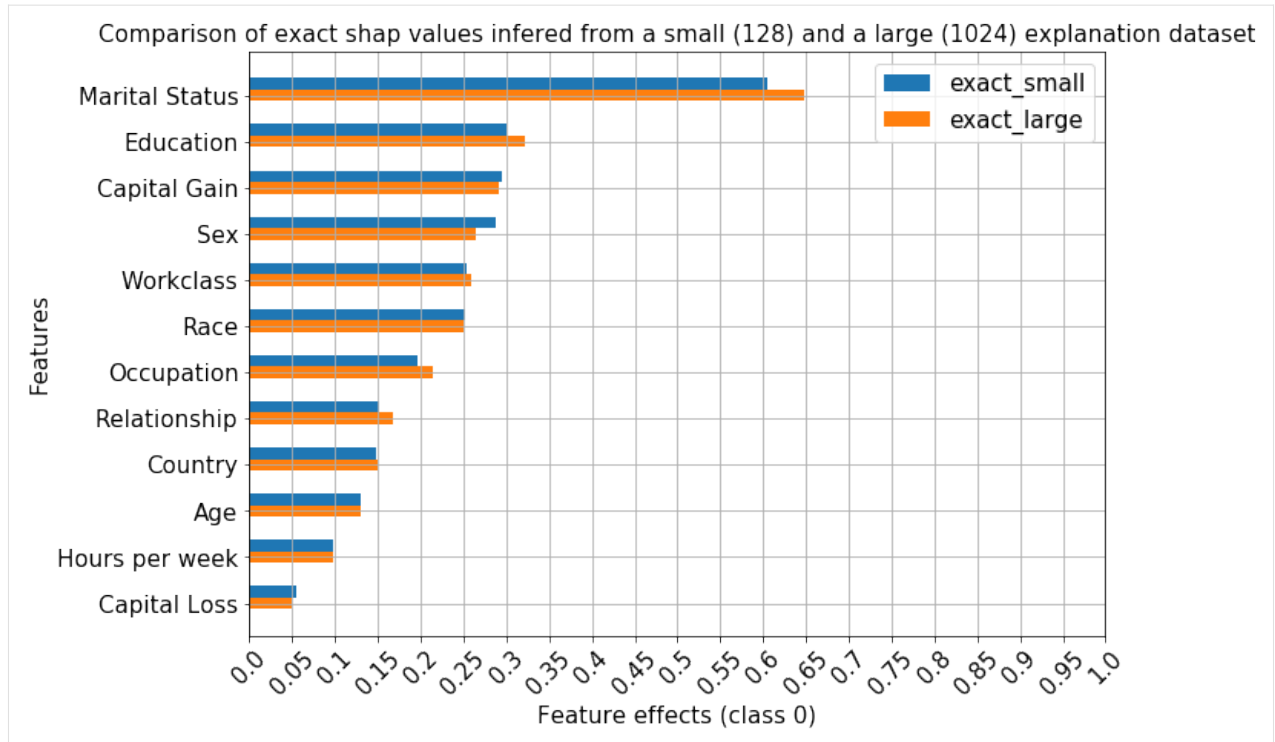
```
[366]: fraction_explained = 0.4
X_explain_large, y_explain_large = split_set(X_test,
                                             y_test,
                                             fraction_explained,
                                             )
X_explain_large_proc = preprocessor.transform(X_explain_large)
X_explain_large_proc_d = sparse2ndarray(X_explain_large_proc)

Number of records: 1024
Number of class 0: 763
Number of class 1: 261

[ ]: data = get_dataset(X_train, y_train, 0.08)
explainer = KernelShap(pred_fcn, link='logit')
explainer.fit(data, group_names=group_names, groups=groups)
explanation_large_dataset = explainer.explain(X_explain_large_proc_d)
ranked_avg_shap_l = get_ranked_values(explanation_large_dataset)

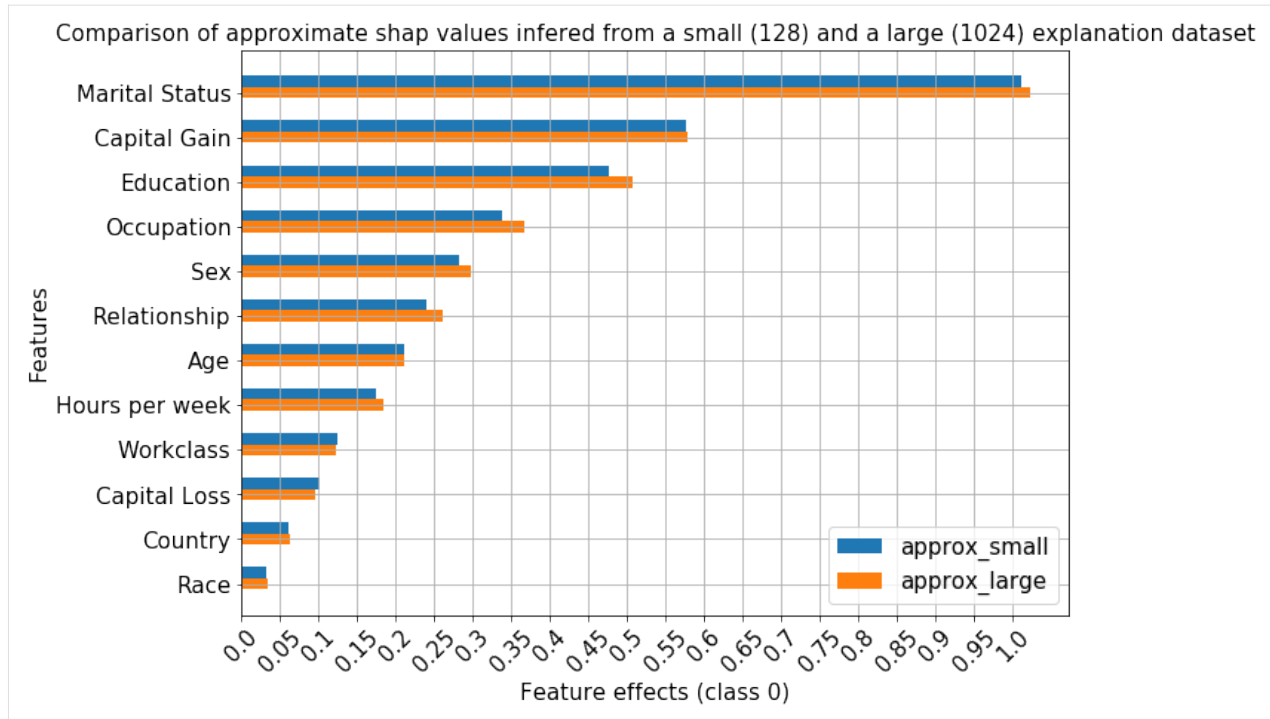
[370]: class_idx = 0 # income below $50,000
exact_shap_large = [(beta[:, None, :]*X_explain_large_proc_d)[i, ...] for i in
                    range(beta.shape[0])]
combined_exact_shap_large = [sum_categories(shap_values, cat_feat_start, feat_enc_dim)
                             for shap_values in exact_shap_large]
ranked_combined_exact_shap_large = [rank_features(shap_values, perm_feat_names) for shap_
                                     values in combined_exact_shap_large]

[383]: comparisons = [ranked_combined_exact_shap]
methods = ['exact_large', 'exact_small']
_, fg, df = compare_avg_mag_shap(class_idx,
                                 comparisons,
                                 ranked_combined_exact_shap_large,
                                 methods=methods,
                                 bar_width=0.5,
                                 legend_fontsize=15,
                                 axes_label_fontsize=15,
                                 tick_labels_fontsize=15,
                                 title="Comparison of exact shap values inferred from a
                                     ↪ small (128) and a large (1024) explanation dataset",
                                 title_fontsize=15,
                                 xlabel=f'Feature effects (class {class_idx})',
                                 ylabel='Features'
                                 )
```



As expected, the exact shap values have the same ranking when a larger set is explained, since they are derived from the same model coefficients.

```
[387]: comparisons = [ranked_avg_shap]
methods = ['approx_large', 'approx_small']
_, fg, df = compare_avg_mag_shap(class_idx,
                                comparisons,
                                ranked_avg_shap_1,
                                methods=methods,
                                bar_width=0.5,
                                legend_fontsize=15,
                                axes_label_fontsize=15,
                                tick_labels_fontsize=15,
                                title="Comparison of approximate shap values inferred
↪ from a small (128) and a large (1024) explanation dataset",
                                title_fontsize=15,
                                xlabel=f'Feature effects (class {class_idx})',
                                ylabel='Features'
                                )
```



The ranking of the features also remains unchanged for the approximate method even when significantly more instances are explained.

```
[389]: with open('large_explain_set.pkl', 'wb') as f:
        pickle.dump(
            {'data': data,
             'explainer': explainer,
             'raw_shap': explanation_large_dataset,
             'ranked_shap_vals': ranked_avg_shap_1
            },
            f
        )
```

Footnotes

(1): As detailed in *Theorem 1* in [3], the estimation process for a shap value of feature i from instance x involves taking a weighted average of the contribution of feature i to the model output, where the weighting takes into account all the possible orderings in which the previous and successor features can be added to the set. This computation is thus performed by choosing subsets of features from the full feature set and setting the values of these features to a *background value*; the prediction on these perturbed samples is used in a least squares objective (*Theorem 2*), weighted by the Shapley kernel. Note that the optimisation objective involves a summation over all possible subsets. Enumerating all the feature subsets has exponential computational cost, so the smaller the feature set, the more samples can be drawn and more accurate shap values can be estimated. Thus, grouping the features can serve to reduce the variance of the shap values estimation by providing a smaller set of features to choose from.

(2): This is a kwarg to `shap_values` method.

(3): Note the progress bars below show, however, different runtimes between the two methods. No accurate timing analysis was carried out to study this aspect.

(4): Note that the `shap` library currently does not support grouping when the data is represented as a sparse matrix, so it should be converted to a `numpy.ndarray` object, both during explainer initialisation and when calling the `shap_values` method.

(5): When `link='logit'` is passed to the plotting function, the model outputs are scaled to the probability space, so the *inverse logit transformation* is applied to the data and axis ticks. This is in contrast to passing `link='logit'` to the `KernelExplainer`, which maps the model output through the *forward logit transformation*, $\log\left(\frac{p}{1-p}\right)$.

(6): We could alter the base value by specifying the `new_base_value` argument to `shap.decision_plot`. Note that this argument has to be specified in the *same* units as the explanation - if we explained the instances in margin space then to switch the base value of the plot to, say, $p=0.4$ then we would pass `new_base_value = log(0.4/(1 - 0.4))` to the plotting function.

(7): In this context, bias refers to the bias-variance tradeoff; a simpler model will likely incur a larger error during training but will have a smaller generalisation gap compared to a more complex model which will have smaller training error but will generalise poorly.

References

- [1] Mahto, K.K., 2019. “One-Hot-Encoding, Multicollinearity and the Dummy Variable Trap”. Retrieved 02 Feb 2020 ([link](#))
- [2] Mood, C., 2017. “Logistic regression: Uncovering unobserved heterogeneity.”
- [3] Lundberg, S.M. and Lee, S.I., 2017. A unified approach to interpreting model predictions. In *Advances in neural information processing systems* (pp. 4765-4774).
- [4] Lundberg, S.M., Erion, G., Chen, H., DeGrave, A., Prutkin, J.M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N. and Lee, S.I., 2020. From local explanations to global understanding with explainable AI for trees. *Nature machine intelligence*, 2(1), pp.56-67.
- [5] Lundberg, S.M., Nair, B., Vavilala, M.S., Horibe, M., Eisses, M.J., Adams, T., Liston, D.E., Low, D.K.W., Newman, S.F., Kim, J. and Lee, S.I., 2018. Explainable machine-learning predictions for the prevention of hypoxaemia during surgery. *Nature biomedical engineering*, 2(10), pp.749-760.

8.9.4 Kernel SHAP explanation for SVM models

Note

To enable SHAP support, you may need to run

```
pip install alibi[shap]
```

```
[ ]: # shap.summary_plot currently doesn't work with matplotlib>=3.6.0,  
# see bug report: https://github.com/slundberg/shap/issues/2687  
!pip install matplotlib==3.5.3
```


Introduction

In this example, we show how to explain a multi-class classification model based on the SVM algorithm using the KernelSHAP method. We show how to perform instance-level (or *local*) explanations on this model as well as how to draw insights about the model behaviour in general by aggregating information from explanations across many instances (that is, perform *global explanations*).

```
[ ]: import shap
shap.initjs()

import matplotlib.pyplot as plt
import numpy as np

from alibi.explainers import KernelShap
from sklearn import svm
from sklearn.datasets import load_wine
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
```

Data preparation

```
[2]: wine = load_wine()
wine.keys()

[2]: dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names'])

[3]: data = wine.data
target = wine.target
target_names = wine.target_names
feature_names = wine.feature_names
```

Split data into testing and training sets and normalize it.

```
[4]: X_train, X_test, y_train, y_test = train_test_split(data,
                                                         target,
                                                         test_size=0.2,
                                                         random_state=0,
                                                         )

print("Training records: {}".format(X_train.shape[0]))
print("Testing records: {}".format(X_test.shape[0]))

Training records: 142
Testing records: 36

[5]: scaler = StandardScaler().fit(X_train)
X_train_norm = scaler.transform(X_train)
X_test_norm = scaler.transform(X_test)
```

Fitting a support vector classifier (SVC) to the Wine dataset

Training

SVM, is a binary classifier, so multiple classifiers are fitted in order to support multiclass classification. The algorithm output is explained here.

```
[6]: np.random.seed(0)
      classifier = SVC(
          kernel = 'rbf',
          C=1,
          gamma = 0.1,
          decision_function_shape='ovr', # n_cls trained with data from one class as positive,
          ↪and remainder of data as neg
          random_state = 0,
      )
      classifier.fit(X_train_norm, y_train)

[6]: SVC(C=1, gamma=0.1, random_state=0)
```

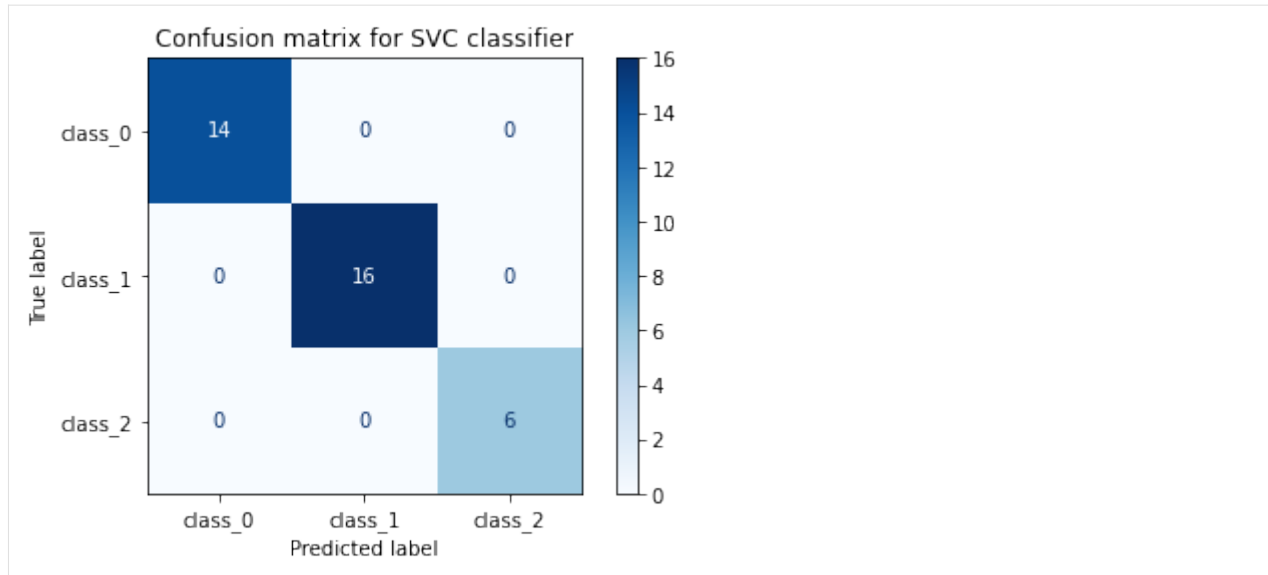
Model assessment

Look at confusion matrix.

```
[7]: y_pred = classifier.predict(X_test_norm)

[8]: cm = confusion_matrix(y_test, y_pred)

[9]: title = 'Confusion matrix for SVC classifier'
      disp = ConfusionMatrixDisplay.from_estimator(classifier,
          X_test_norm,
          y_test,
          display_labels=target_names,
          cmap=plt.cm.Blues,
          normalize=None,
          )
      disp.ax_.set_title(title);
```



The confusion matrix shows the classifier is perfect - let's understand what patterns in the data help the SVC perform so well!

Apply KernelSHAP to explain the model

The model needs access to a function that takes as an input samples and returns predictions to be explained. For an input z the decision function of a *binary* SVM classifier is given by:

$$\text{class}(z) = \text{sign}(\beta z + b)$$

where β is the best separating hyperplane (linear combination of support vectors, the training points closest to the separating hyperplane) and b is the bias of the model.

For the 'one-vs-rest' SVM, `nclass` binary SVM algorithms are fitted using each class as the positive class and the remainder as negative class. The classification decision is taken by assigning the label from the classifier with the maximum absolute decision score. Therefore, to explain our model we could consider explaining the SVM model which outputs the highest decision score. [Click here to go back to source.](#)

To do so, the KernelSHAP explainer must receive a callable that returns a set of scores when called with an input X , in this case the `decision_function` attribute of our classifier.

```
[10]: pred_fcn = classifier.decision_function
```

```
[11]: np.random.seed(0)
svm_explainer = KernelShap(pred_fcn)
svm_explainer.fit(X_train_norm)
```

Using 142 background data samples could cause slower run times. Consider using `shap.sample(data, K)` or `shap.kmeans(data, K)` to summarize the background as K samples.

```
[11]: KernelShap(meta={
    'name': 'KernelShap',
    'type': ['blackbox'],
    'task': 'classification',
    'explanations': ['local', 'global'],
    'params': {
```

(continues on next page)

(continued from previous page)

```

        'link': 'identity',
        'group_names': None,
        'grouped': False,
        'groups': None,
        'weights': None,
        'summarise_background': False,
        'summarise_result': None,
        'transpose': False,
        'kwargs': {}

    'version': '0.7.1dev'
)

```

Note that the explainer is fit to the classifier training set. This training set is used for two purposes:

- To determine the model output when all inputs are missing (ϕ_0 in eq. (5) of [1]). Because the SVM model does not accept arbitrary inputs, this quantity is approximated by averaging the decision score for each class, across the samples in `X_train_norm` as shown below and it is stored as the `expected_value` attribute of the explainer
- The values of the features in the $N \times D$ `X_train_norm` dataset are used to replace the values missing during the feature attribution (ϕ_i) estimation process. Specifically, `nsamples` copies of `X_train_norm` are tiled to create a dataset where, for each copy, a subset of features z' of size $s = |z'|$ are replaced by the values in the instance to be explained and the complement of this subset is left to the background dataset value. These background values simulate the effect of *missing values*, since most models cannot cope with arbitrary patterns of missing values at inference time. Therefore, when computing the shap value of a particular feature, ϕ_i , `nsamples` regression targets ($f(h_x(z'))$ in eq (5) of [1]) are computed as the *expected* prediction of the model to be explained when a given subset of features is missing as opposed to replacing the missing feature with a single value. Note that the averaging operation can be replaced by weighted averaging by specifying the `weights` argument to the `fit` method. (a)

For the above reason, this is sometimes referred to as the *background dataset*; a larger dataset increases the runtime of the algorithm, so for large datasets, a subset of it should be used. An option to deal with the runtime issue while still providing meaningful values for missing values is to summarise the dataset using the `shap.kmeans` function. This function wraps the `sklearn` k-means clustering implementation, while ensuring that the clusters returned have values that are found in the training data. In addition, the samples are weighted according to the cluster sizes.

```

[12]: # expected_values attribute stores average scores across training set for every binary_
      ↪ SVM
      mean_scores_train = pred_fcn(X_train_norm).mean(axis=0)
      # are stored in the expected value attribute of the explainer ...
      print(mean_scores_train - svm_explainer.expected_value)

[-1.11022302e-16  4.44089210e-16 -4.44089210e-16]

```

```

[13]: svm_explanation = svm_explainer.explain(X_test_norm, l1_reg=False)

0%|          | 0/36 [00:00<?, ?it/s]

```

In cases where the feature space has higher dimensionality, only a small fraction of the missing subsets can be enumerated for a given number of samples `nsamples`. If the the fraction of the subsets enumerated falls below a fraction (0.2 for version 0.3.2) and the regularisation is set to `auto`, a least angle regression with the AIC information criterion for selecting the regularisation coefficient α is performed in order to select features. The regularisation has no effect if the fraction is greater than this threshold and `l1_reg` is not set to `auto`. Other options for regularisations are: - `l1_reg="num_features(10)"`: in this case, the LARS algorithm [2] is used to which 10 features to estimate the shap values for. - `l1_reg="bic"`: in this case, the least angle regression is run with the Bayes Information Criterion -

`l1_reg=0.02`: if a float is specified, the ℓ_1 -regularised regression coefficient is set to this value

Local explanation

Because the SVM algorithm returns a score for each of the 3 classes, the `shap_values` are computed for each class in turn. Moreover, the attributions are computed for each data point to be explained and for each feature, resulting in a $N_e \times D$ matrix of shap values for each class, where N_e is the number of instances to be explained and D is the number of features.

```
[14]: print("Output type:", type(svm_explanation.shap_values))
      print("Output size:", len(svm_explanation.shap_values))
      print("Class output size:", svm_explanation.shap_values[0].shape)
```

```
Output type: <class 'list'>
Output size: 3
Class output size: (36, 13)
```

For a given instance, we can visualise the attributions using a force plot. Let's choose the first example in the testing set as an example.

```
[15]: idx = 0
      instance = X_test_norm[idx][None, :]
      pred = classifier.predict(instance)
      scores = classifier.decision_function(instance)
      class_idx = pred.item()
      print("The predicted class for the X_test_norm[{}] is {}".format(idx, *pred))
      print("OVR decision function values are {}".format(*scores))
```

```
The predicted class for the X_test_norm[0] is 0.
OVR decision function values are [ 2.24071294  0.85398239 -0.21510456].
```

We see that class 0 is predicted because the SVM model trained with class 0 as a positive class and classes 1 and 2 combined as a negative class returned the largest score.

To create this force plot, we have provided the plotting function with four inputs: - the expected predicted score by the class-0 SVM assuming all inputs are missing. This is marked as the base value on the force plot - the feature attributions for the instance to be explained - the instance to be explained - the feature names

```
[16]: shap.force_plot(
      svm_explainer.expected_value[class_idx],
      svm_explanation.shap_values[class_idx][idx, :] ,
      instance,
      feature_names,
      )
```

```
[16]: <shap.plots._force.AdditiveForceVisualizer at 0x7f1ddad28070>
```

The force plot depicts the contribution of each feature to the process of moving the value of the decision score from the *base value* (estimation of the decision score if all inputs were missing) to the value predicted by the classifier. We see that all features contribute to increasing the decision score, and that the largest increases are due to the `proline` feature with a value of 1.049 and the `flavanoids` feature with a value of 0.9778. The lengths of the bars are the corresponding feature attributions.

Similarly, below we see that the `proline` and `alcohol` features contribute to decreasing the decision score of the SVM predicting class 1 as positive and that the `malic_acid` feature increases the decision score.

```
[17]: shap.force_plot(
        svm_explainer.expected_value[1],
        svm_explanation.shap_values[1][idx, :] ,
        instance,
        feature_names,
    )

[17]: <shap.plots._force.AdditiveForceVisualizer at 0x7f1ddad28a90>
```

An alternative way to visualise local explanations for multi-output models is a *multioutput decision plot*. This plot can be especially useful when the number of features is large and the force plot might not be readable.

```
[18]: def class_labels(classifier, instance, class_names=None):
        """
        Creates a set of legend labels based on the decision
        scores of a classifier and, optionally, the class names.
        """

        decision_scores = classifier.decision_function(instance)

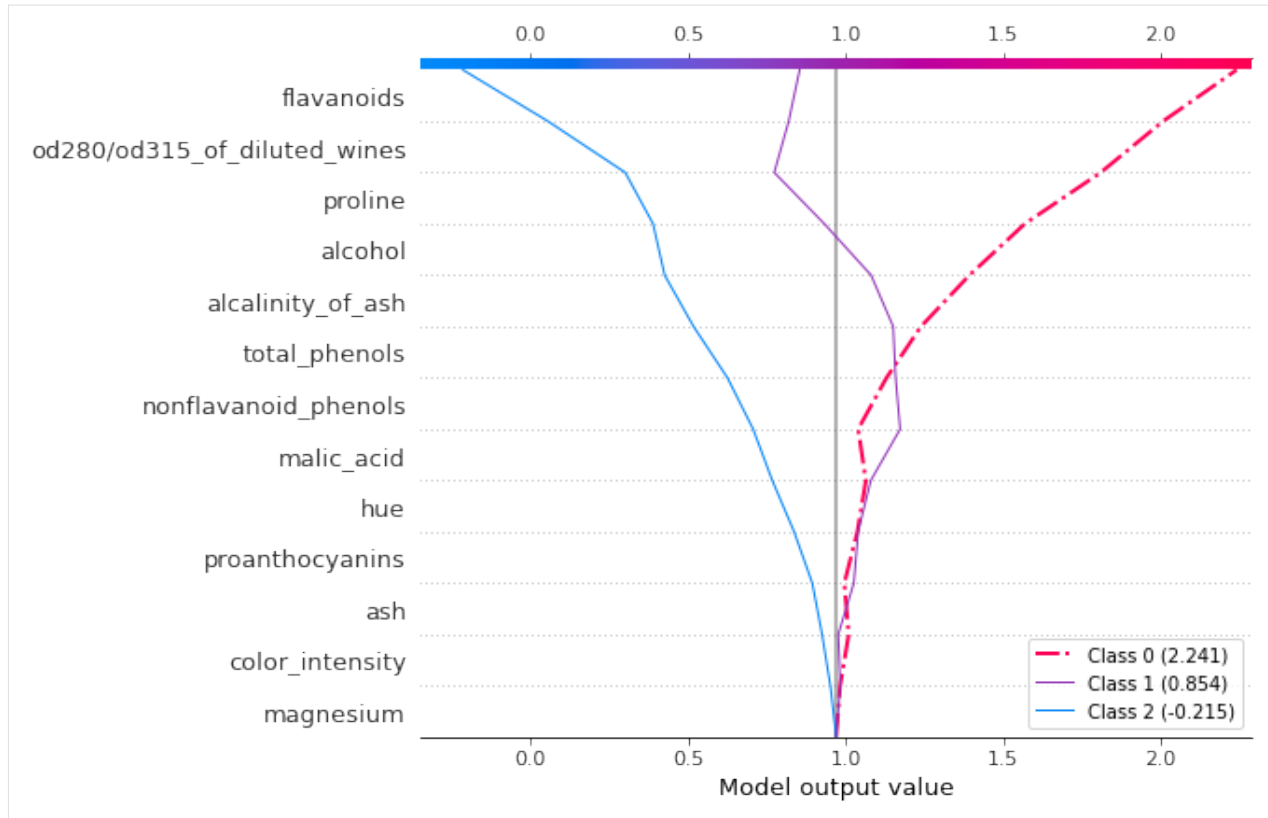
        if not class_names:
            class_names = [f'Class {i}' for i in range(decision_scores.shape[1])]

        for i, score in enumerate(np.nditer(decision_scores)):
            class_names[i] = class_names[i] + ' ({}).format(round(score.item(),3))

        return class_names

[19]: legend_labels = class_labels(classifier, instance)

[20]: r = shap.multioutput_decision_plot(svm_explainer.expected_value.tolist(),
        svm_explanation.shap_values,
        idx,
        feature_names=feature_names,
        feature_order='importance',
        highlight=[class_idx],
        legend_labels=legend_labels,
        return_objects=True,
        legend_location='lower right')
```



The decision plots shows how the individual features influence contribute to the classification into each of the three classes (a *prediction path*). One sees that, for this example, the model can easily separate the three classes. It also shows, for example, that a wine with the given alcohol content is typical of class 0 (because the alcohol feature contributes positively to a classification of class 0 as negatively to classification in classes 1 and 2)

Note that the feature ordering is determined by summing the shap value magnitudes corresponding to each feature across classes and then ordering the `feature_names` in descending order of cumulative magnitude. The plot origin, marked by the gray vertical line, is the average base values across the classes. The dashed line represents the model prediction - in general we can highlight a particular class by passing the class index in the `highlight` list.

Suppose now that we want to analyse instance 5 but realise that the feature importances are different for this instance.

```
[21]: idx = 5
      instance = X_test_norm[idx][None, :]
      pred = classifier.predict(instance)
```

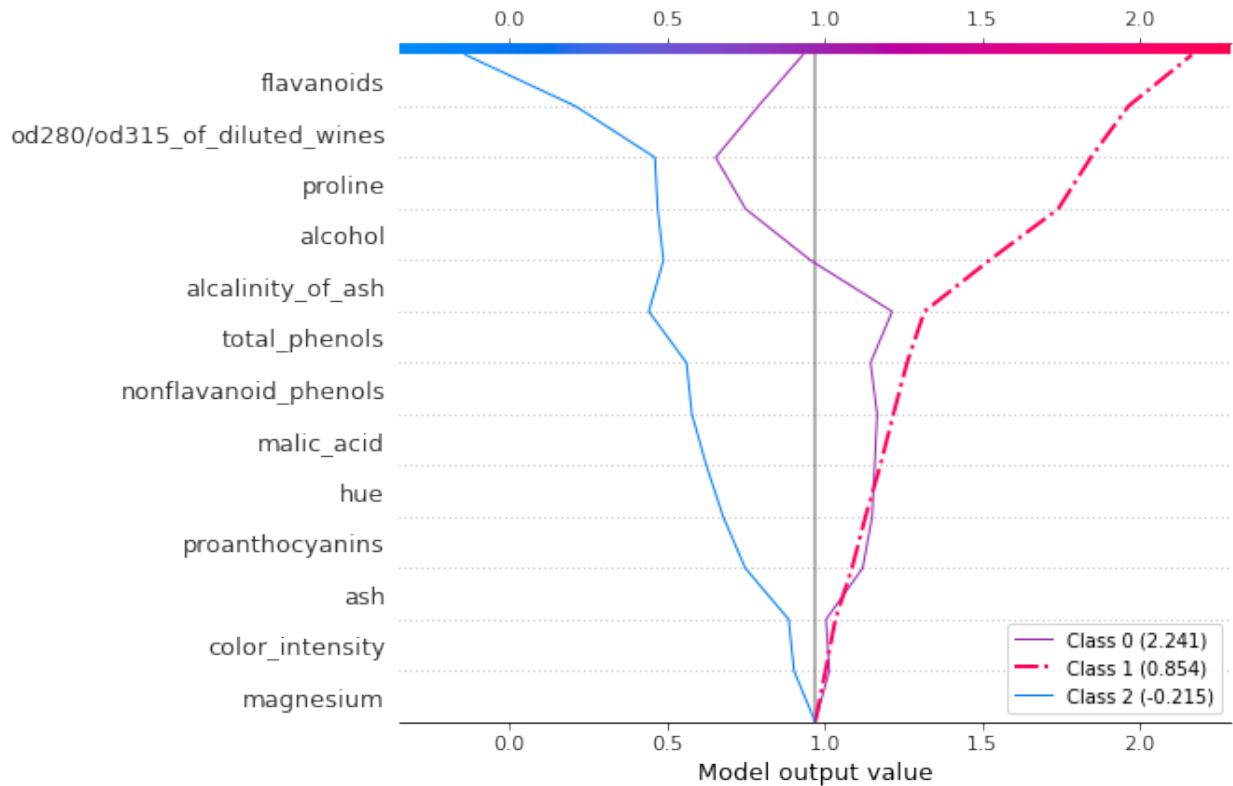
```
[22]: instance_shap = np.array(svm_explanation.shap_values)[: , idx, :]
      feature_order = np.argsort(np.sum(np.abs(instance_shap), axis=0))[:, :-1]
      feat_importance = [feature_names[i] for i in feature_order]
```

```
[23]: print(feat_importance)

['flavanoids', 'alkalinity_of_ash', 'od280/od315_of_diluted_wines', 'alcohol', 'ash',
 → 'total_phenols', 'proline', 'magnesium', 'proanthocyanins', 'hue', 'malic_acid',
 → 'nonflavanoid_phenols', 'color_intensity']
```

We want to create a multi-output decision plot with the same feature order and scale. This is possible, since by passing the `return_objects=True` to the plotting function in the example above, we retrieved the feature indices and the axis limits and can reuse them to display the decision plot for instance 5 with the same feature order as above.

```
[24]: shap.multioutput_decision_plot(svm_explainer.expected_value.tolist(),
                                   svm_explanation.shap_values,
                                   idx,
                                   feature_names=feature_names,
                                   feature_order=r.feature_idx,
                                   highlight=[pred.item()],
                                   legend_labels=legend_labels,
                                   legend_location='lower right',
                                   xlim=r.xlim,
                                   return_objects=False)
```



Global explanation

As shown above, the force plot allows us to understand how the individual features contribute to a classification output *given* an instance. However, the particular explanation does not tell us about the model behaviour in general. Below, we show how such insights can be drawn.

Stacked force plots

The simplest way we can do this is to stack the force plot for a number of instances, which can be achieved by calling the `force_plot` function with the same arguments as before but replacing `instance` with the whole testing set, `X_test_norm`.

```
[25]: class_idx = 0 # we explain the predicted label
      shap.force_plot(
          svm_explainer.expected_value[class_idx],
          svm_explanation.shap_values[class_idx],
          X_test_norm,
          feature_names,
      )
```

```
[25]: <shap.plots._force.AdditiveForceArrayVisualizer at 0x7f1dda9f2dc0>
```

In the default configuration, the x axis is represented by the 36 instances in `X_test_norm` whereas the y axis represents the decision score. Note that, like before, the decision score is for class 0. For a given instance, the height in between two horizontal lines is equal to the shap value of the feature, and hovering over a plot shows a list of the features along with their values, sorted by shap values. As before, the blue shading shows a negative contribution to the decision score (moves the score *away* from the baseline value) whereas the pink shading shows a positive contribution to the decision score. Hovering over the plot, tells us, for example, that to achieve a high decision score (equivalent to class 0 membership) the features `proline` and `flavanoids` are generally the most important and that positive `proline` values lead to higher decision scores for belonging to this class whereas negative `proline` values provide evidence against this one belonging to 0 class.

To see the relationship between decision scores and the values more clearly, we can permute the x axis so that the instances are sorted according to the value of the `proline` feature by selecting `proline` from the horizontal drop down menu.

```
[26]: shap.force_plot(
      svm_explainer.expected_value[class_idx],
      svm_explanation.shap_values[class_idx],
      X_test_norm,
      feature_names,
  )
```

```
[26]: <shap.plots._force.AdditiveForceArrayVisualizer at 0x7f1dda98bfd0>
```

You can also explore the effect of a particular feature across the testing dataset. For example, in the plot below, by selecting `flavanoids` from the top drop-down, the instances are ordered on the x axis in increasing value of the `flavanoids` feature.

Similarly, selecting `flavanoids effects` from the side drop-down will plot the shap value as opposed to the model output. The effect of this feature generally increases as its value increases and the large negative values of this feature reduce the decision score for classification as 0. Note that the shap values are resrepresented with respect to the base value for this class (0.798, as shown below).

```
[27]: shap.force_plot(
      svm_explainer.expected_value[0],
      svm_explanation.shap_values[0],
      X_test_norm,
      feature_names,
  )
```

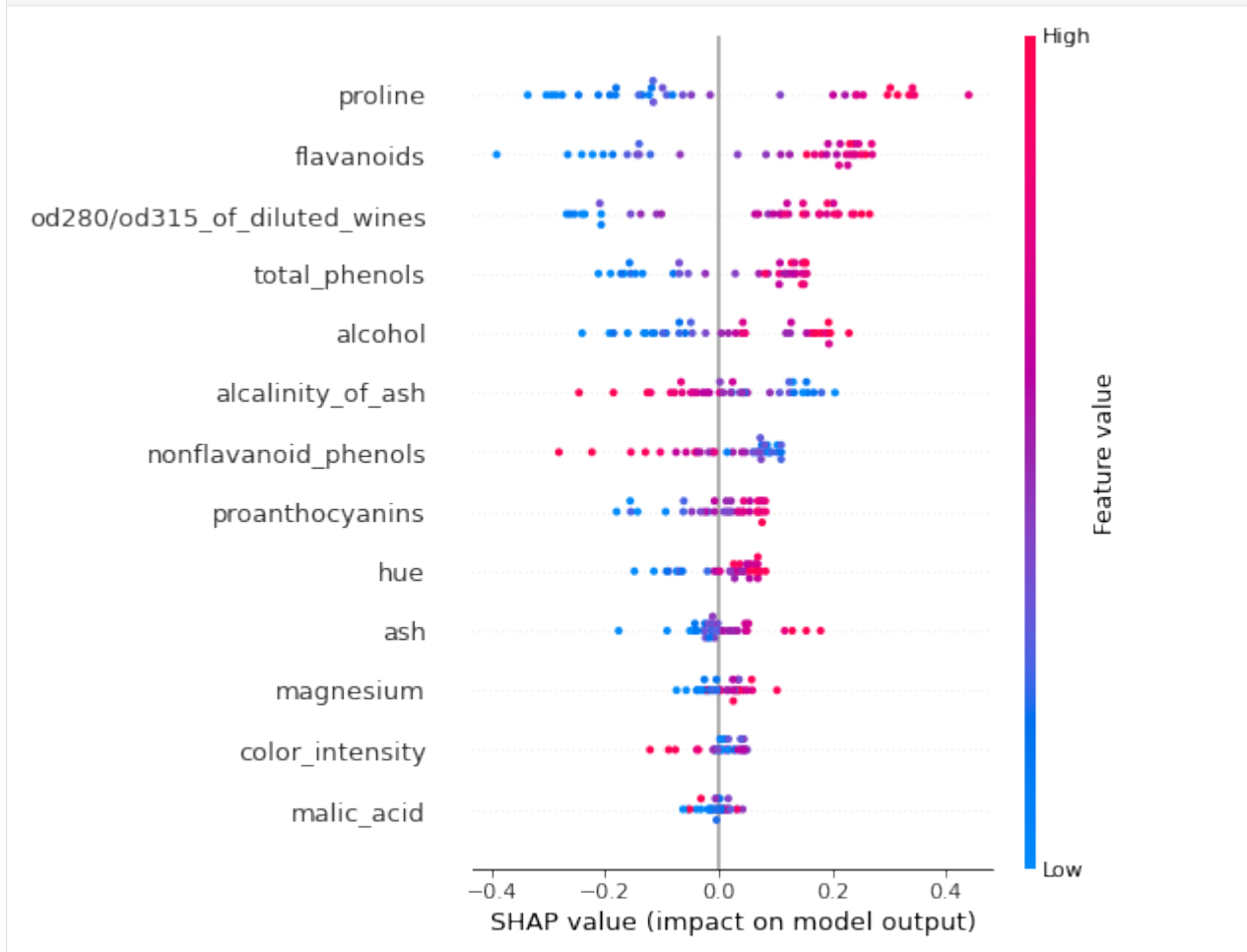
```
[27]: <shap.plots._force.AdditiveForceArrayVisualizer at 0x7f1dda8faac0>
```

```
[28]: print(svm_explainer.expected_value)
[0.79821894 1.41710253 0.69461514]
```

Summary plots

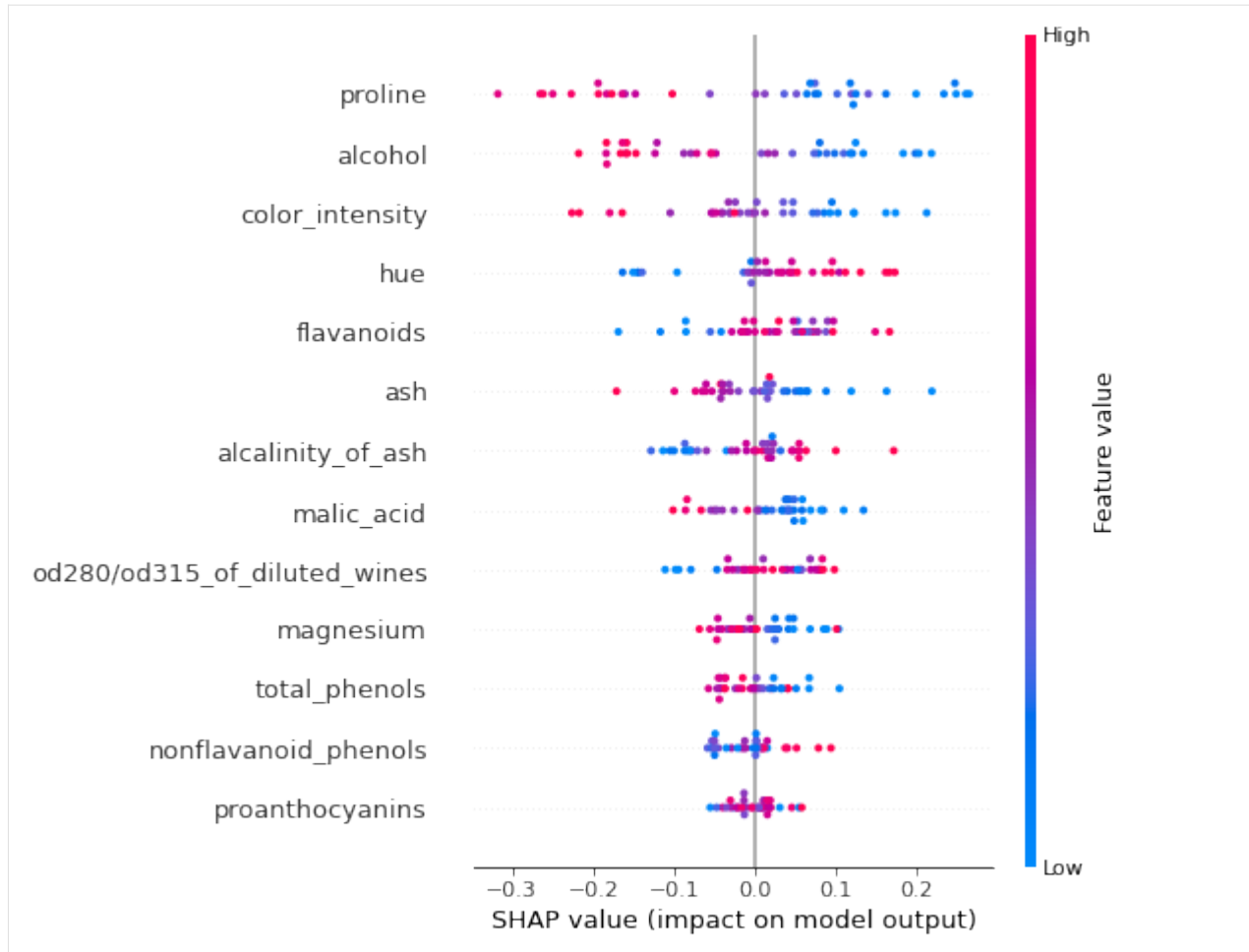
To visualise the impact of the features on the decision scores associated with class 0, we can use a summary plot. In this plot, the features are sorted by the sum of their SHAP values magnitudes across all instances in `X_test_norm`. Therefore, the features with the highest impact on the decision score for class `class_idx` are displayed at the top of the plot.

```
[29]: shap.summary_plot(svm_explanation.shap_values[0], X_test_norm, feature_names)
```



In this case, the `proline` and `flavanoids` have the most impact on the model output; as the values of the features increase, their impact also increases and the model is more likely to predict class 0. On the other hand, high values of the `nonflavanoid_phenols` have a negative impact on the model output, potentially contributing to the classification of the particular wine in a different class. To see this, we do a summary plot with respect to `class_2`.

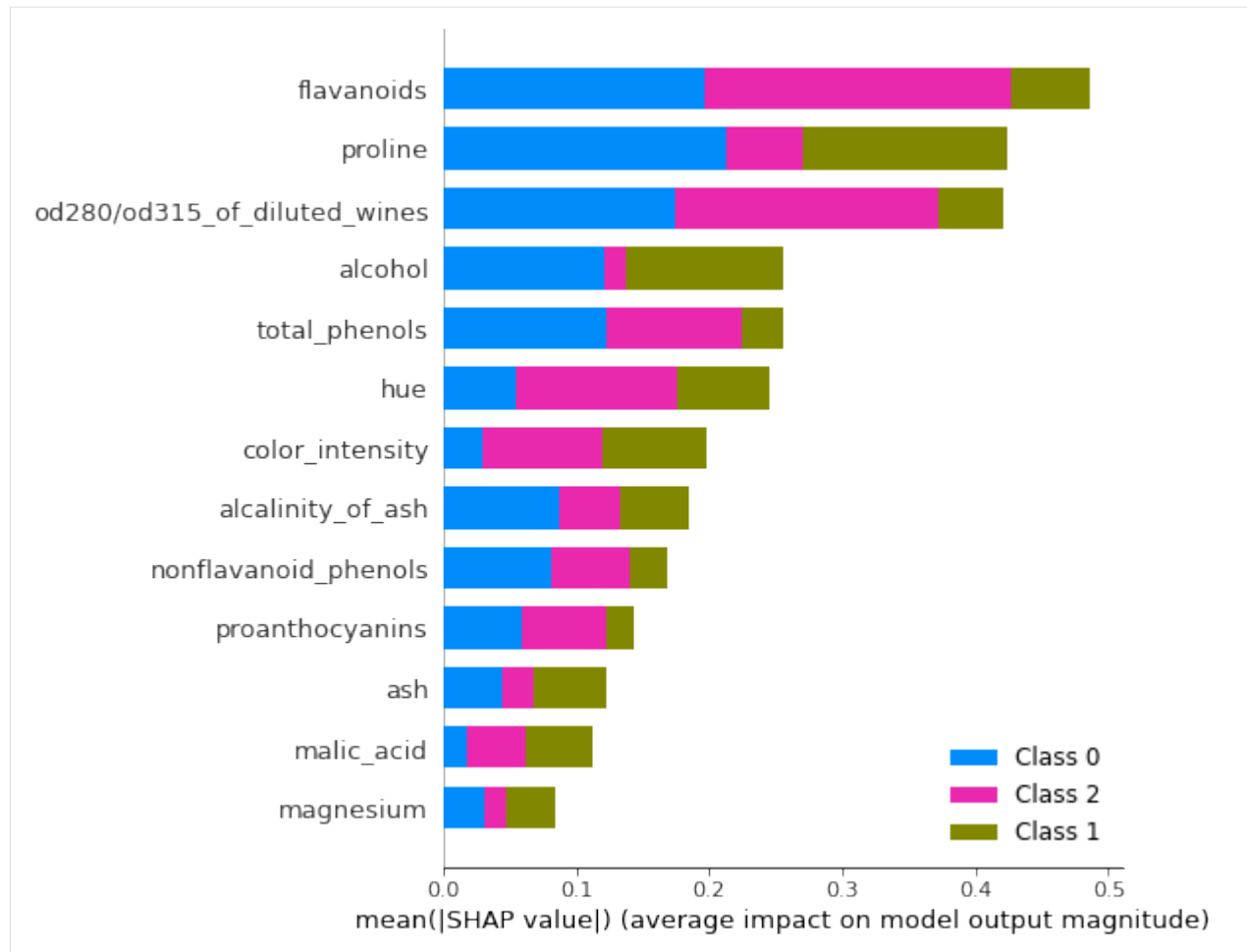
```
[30]: shap.summary_plot(svm_explanation.shap_values[1], X_test_norm, feature_names)
```



We see that, indeed, a higher value of the `nonflavanoid_phenols` feature contributes to a sample being classified as class 1, but that this effect is rather limited compared to features such as `proline` or `alcohol`.

To visualise the impact of the feature across all classes, that is, the importance of a particular feature for the model, we simply pass all the shap values to the `summary_plot` functions. We see, that, for example, the `color_intensity` feature is much more important for deciding whether an instance should be classified as `class_2` then in `class_0`.

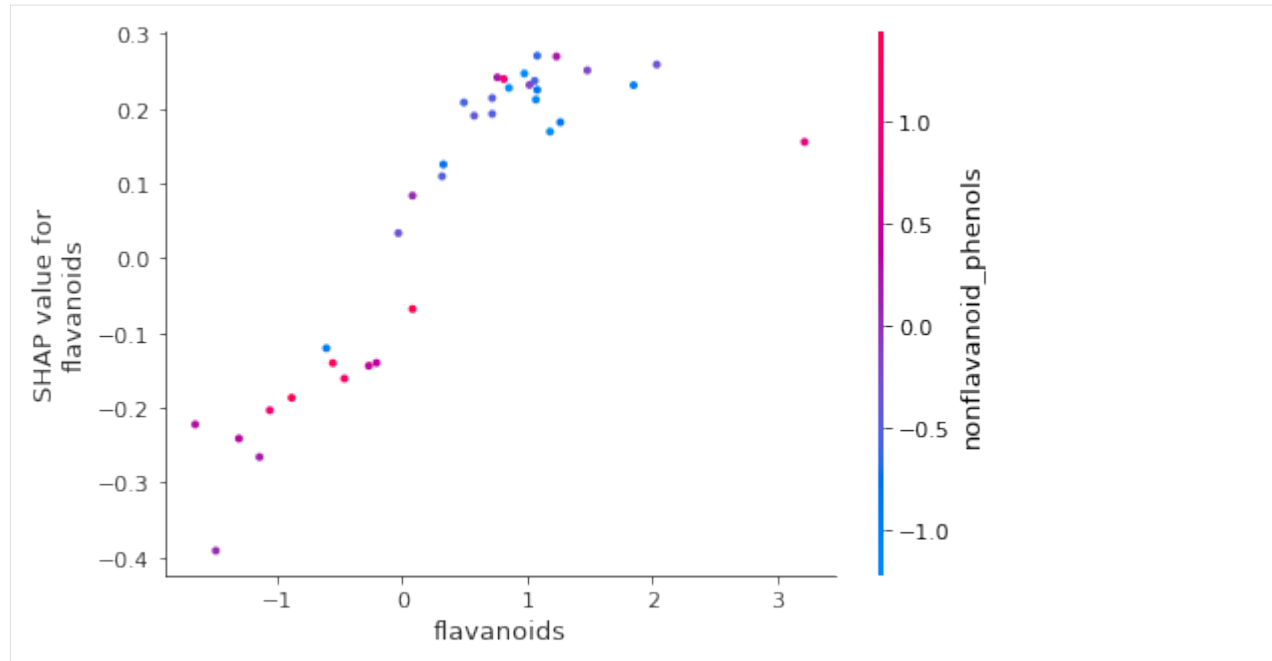
```
[31]: shap.summary_plot(svm_explanation.shap_values, X_test_norm, feature_names)
```



Dependence plots

Another way to visualise the model dependence on a particular feature is through a dependence plot. This plot shows the impact of the feature value on its importance for classification with respect to class 0.

```
[32]: feature = 'flavanoids'
      shap.dependence_plot(
          feature,
          svm_explanation.shap_values[0],
          X_test_norm,
          feature_names=feature_names,
          interaction_index='auto',
      )
```



The colour of the individual instances is represented by the value of the feature `nonflavanoid_phenols`. By specifying `interaction_index=auto`, the `nonflavanoid_phenols` was estimated as the feature with the strongest interaction with the `flavanoids_feature`; this interaction is approximate, and is estimated by computing the Pearson Correlation Coefficient between the shap values of the reference feature (`flavanoids` in this case) and the value of each feature in turn on bins along the feature value.

We see that, for class 0 wines, a higher value for `nonflavanoid_phenols` is generally associated with a low value in `flavanoids` and that they have a negative impact on the score for class 0 classification.

Footnotes

(a) The weights are applied to each point in a copy, so the number of weights should be the same as the number of samples in the data.

References

- [1] Lundberg, S.M. and Lee, S.I., 2017. A unified approach to interpreting model predictions. In Advances in neural information processing systems (pp. 4765-4774).
- [2] Wikipedia entry on the Least-angle regression: https://en.wikipedia.org/wiki/Least-angle_regression.

8.9.5 Kernel SHAP explanation for multinomial logistic regression models

Note

To enable SHAP support, you may need to run

```
pip install alibi[shap]
```

```
[ ]: # shap.summary_plot currently doesn't work with matplotlib>=3.6.0,  
# see bug report: https://github.com/slundberg/shap/issues/2687  
!pip install matplotlib==3.5.3
```

Introduction

In a previous *example*, we showed how the KernelSHAP algorithm can be applied to explain the output of an arbitrary classification model so long the model outputs probabilities or operates in margin space. We also showcased the powerful visualisations in the shap library that can be used for model investigation. In this example we focus on understanding, in a simple setting, how conclusions drawn from the analysis of the KernelShap output relate to conclusions drawn from interpreting the model directly. To make this possible, we fit a logistic regression model on the Wine dataset.

```
[ ]: import shap  
shap.initjs()  
  
import matplotlib.pyplot as plt  
import numpy as np  
  
from alibi.explainers import KernelShap  
from scipy.special import logit  
from sklearn.datasets import load_wine  
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay  
from sklearn.model_selection import train_test_split  
from sklearn.preprocessing import StandardScaler  
from sklearn.linear_model import LogisticRegression
```

Data preparation: load and split Wine dataset

```
[2]: wine = load_wine()  
wine.keys()  
  
[2]: dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names'])  
  
[3]: data = wine.data  
target = wine.target  
target_names = wine.target_names  
feature_names = wine.feature_names
```

Split data into testing and training sets and normalize it.

```
[4]: X_train, X_test, y_train, y_test = train_test_split(data,
                                                    target,
                                                    test_size=0.2,
                                                    random_state=0,
                                                    )
print("Training records: {}".format(X_train.shape[0]))
print("Testing records: {}".format(X_test.shape[0]))

Training records: 142
Testing records: 36
```

```
[5]: scaler = StandardScaler().fit(X_train)
X_train_norm = scaler.transform(X_train)
X_test_norm = scaler.transform(X_test)
```

Fitting a multinomial logistic regression classifier to the Wine dataset

Training

```
[6]: classifier = LogisticRegression(multi_class='multinomial',
                                    random_state=0,
                                    )
classifier.fit(X_train_norm, y_train)

[6]: LogisticRegression(multi_class='multinomial', random_state=0)
```

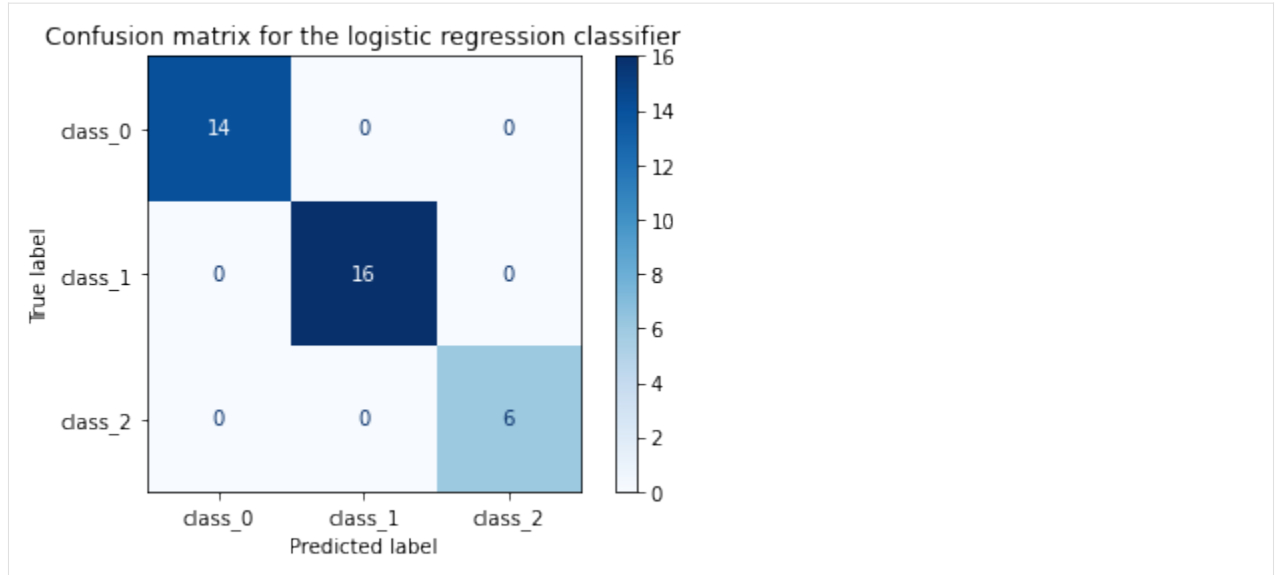
Model assessment

```
[7]: y_pred = classifier.predict(X_test_norm)

[8]: cm = confusion_matrix(y_test, y_pred)

[9]: title = 'Confusion matrix for the logistic regression classifier'
disp = ConfusionMatrixDisplay.from_estimator(classifier,
                                             X_test_norm,
                                             y_test,
                                             display_labels=target_names,
                                             cmap=plt.cm.Blues,
                                             normalize=None,
                                             )

disp.ax_.set_title(title);
```



Interpreting the logistic regression model

One way to arrive at the multinomial logistic regression model is to consider modelling a categorical response variable $y \sim \text{Cat}(y|\beta x)$ where β is $K \times D$ matrix of distribution parameters with K being the number of classes and D the feature dimensionality. Because the probability of outcome k being observed given x , $p_k = p(y = k|x, \beta)$, is bounded by $[0, 1]$, the logistic regression assumes that a linear relationship exists between the *logit* transformation of the output and the input. This can be formalised as follows:

$$\log\left(\frac{p_k}{1 - p_k}\right) = \beta_{0,k} + \beta_{1,k}x_1 + \beta_{2,k}x_2 + \cdots + \beta_{D,k}x_D = \beta_k \cdot x$$

The RHS is a function of the expected value of the categorical distribution (sometimes referred to as a *link function* in the literature). The coefficients β of the linear relations used to fit the logit transformation are estimated jointly given a set of training examples $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$.

For each class, the vector of coefficients β_k can be used to interpret the model *globally*; in the absence of interaction terms, the coefficient of a predictor (i.e., independent variable) represents the *change in log odds* when the predictor changes by one unit while all other variables are kept at fixed values. Equivalently, the exponentiated coefficient is equivalent to a change in odds. Since the transformation from odds to outcome probabilities is monotonic, a change in odds also implies a change in the outcome probability in the same direction. Thus, the magnitudes of the feature coefficients measure the effect of a predictor on the output and thus one can globally interpret the logistic regression model.

However, the log odds ratios and odds ratios are known to be sensitive to *unobserved heterogeneity*, that is, omission of a variable with good explanatory power from a logistic regression model *assumed true*. While we will not be concerned directly with this issue and refer the interested reader to [2], we will be using the *estimated percentage unit effect* (or the *marginal effect*)

$$\beta_{j,k} \times p_{i,k}(1 - p_{i,k})$$

as a means of estimating the effect of a predictor j on individual i ($x_{i,j}$) with respect to predicting the k^{th} class and thus *locally* interpret the model. The average marginal effect is more robust measure of effects in situations where effects are compared across different groups or models. Consider a logistic model where an independent variable x_1 is used to predict an outcome and a logistic model where x_2 , known to be uncorrelated with x_1 , is also included. Since the two models assign different probabilities to the different outcomes and since the distribution of the outcome across values of x_1 should be the same across the two models (due to the independence assumption), we expected the second

model will scale the coefficient of β_1 . Hence, the log-odds and odds ratios are not robust to unobserved heterogeneity so directly comparing the two across models or groups can be misleading. As discussed in [2], the marginal effect is generally robust to the effect.

The average marginal effect (AME) of a predictor

$$\frac{1}{n} \sum_{i=1}^n \beta_{j,k} \times p_{i,k} (1 - p_{i,k})$$

is equivalent to simply using $\beta_{j,k}$ to globally explain the model.

```
[10]: def issorted(arr, reverse=False):
    """
    Checks if a numpy array is sorted.
    """

    if reverse:
        return np.all(arr[::-1][:-1] <= arr[::-1][1:])

    return np.all(arr[:-1] <= arr[1:])

def get_importance(class_idx, beta, feature_names, intercepts=None):
    """
    Retrive and sort abs magnitude of coefficients from model.
    """

    # sort the absolute value of model coef from largest to smallest
    srt_beta_k = np.argsort(np.abs(beta[class_idx, :]))[::-1]
    feat_names = [feature_names[idx] for idx in srt_beta_k]
    feat_imp = beta[class_idx, srt_beta_k]

    # include bias among feat importances
    if intercepts is not None:
        intercept = intercepts[class_idx]
        bias_idx = len(feat_imp) - (np.searchsorted(np.abs(feat_imp)[::-1], np.
↪abs(intercept)))
    #     bias_idx = np.searchsorted(np.abs(feat_imp)[::-1], np.abs(intercept)) + 1
    feat_imp = np.insert(feat_imp, bias_idx, intercept.item(), )
    intercept_idx = np.where(feat_imp == intercept)[0][0]
    feat_names.insert(intercept_idx, 'bias')

    return feat_imp, feat_names

def plot_importance(feat_imp, feat_names, **kwargs):
    """
    Create a horizontal barchart of feature effects, sorted by their magnitude.
    """

    left_x, right_x = kwargs.get("left_x"), kwargs.get("right_x")
    eps_factor = kwargs.get("eps_factor", 4.5)
    xlabel = kwargs.get("xlabel", None)
    ylabel = kwargs.get("ylabel", None)
    labels_fontsize = kwargs.get("labels_fontsize", 15)
    tick_labels_fontsize = kwargs.get("tick_labels_fontsize", 15)
```

(continues on next page)

(continued from previous page)

```

# plot
fig, ax = plt.subplots(figsize=(10, 5))
y_pos = np.arange(len(feats_imp))
ax.barh(y_pos, feats_imp)

# set lables
ax.set_yticks(y_pos)
ax.set_yticklabels(feats_names, fontsize=tick_labels_fontsize)
ax.invert_yaxis() # labels read top-to-bottom
ax.set_xlabel(xlabel, fontsize=labels_fontsize)
ax.set_ylabel(ylabel, fontsize=labels_fontsize)
ax.set_xlim(left=left_x, right=right_x)

# add text
for i, v in enumerate(feats_imp):
    eps = 0.03
    if v < 0:
        eps = -eps_factor*eps
    ax.text(v + eps, i + .25, str(round(v, 3)))

return ax, fig

```

We now retrieve the estimated coefficients, and plot them sorted by their magnitude.

```

[11]: beta = classifier.coef_
intercepts = classifier.intercept_
all_coefs = np.concatenate((beta, intercepts[:, None]), axis=1)

```

```

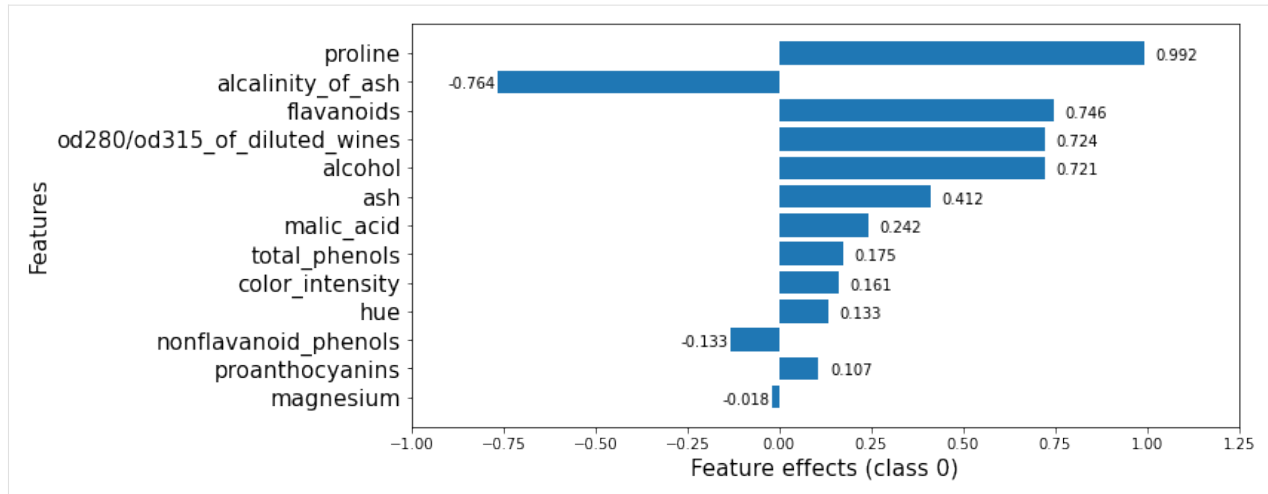
[12]: class_idx = 0
feats_imp, feats_names = get_importance(class_idx,
                                         beta,
                                         feature_names,
                                         )

```

```

[13]: _, class_0_fig = plot_importance(feats_imp,
                                       feats_names,
                                       left_x=-1.,
                                       right_x=1.25,
                                       xlabel = f"Feature effects (class {class_idx})",
                                       ylabel = "Features"
                                       )

```



Note that these effects are with respect to the model bias (displayed below).

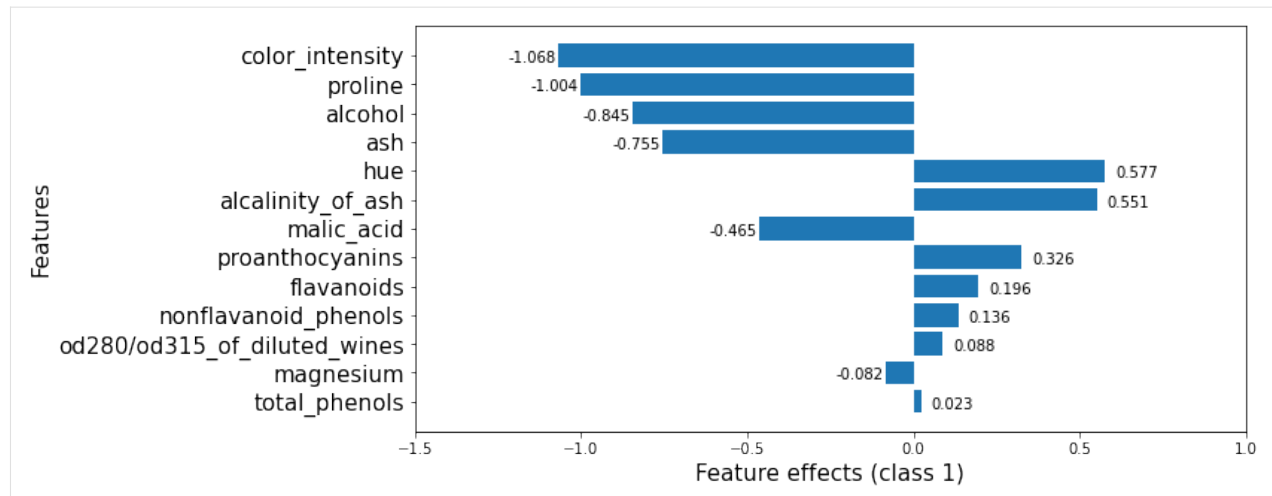
```
[14]: classifier.intercept_
[14]: array([ 0.24013981,  0.66712652, -0.90726633])
```

This plot shows that features such as proline, flavanoids, od280/od315_of_diluted_wines, alcohol increase the odds of any sample being classified as class_0 whereas the alkalinity_of_ash decreases them.

```
[15]: feat_imp, feat_names = get_importance(1,          # class_idx
                                         beta,
                                         feature_names,
                                         )
```

The plot below shows that, however, alkalinity_of_ash increases the odds of a wine being in class_1. Predictors such as proline, alcohol or ash, which increase the odds of predicting a wine as a member of class_0, decrease the odds of predicting it as a member of class_1.

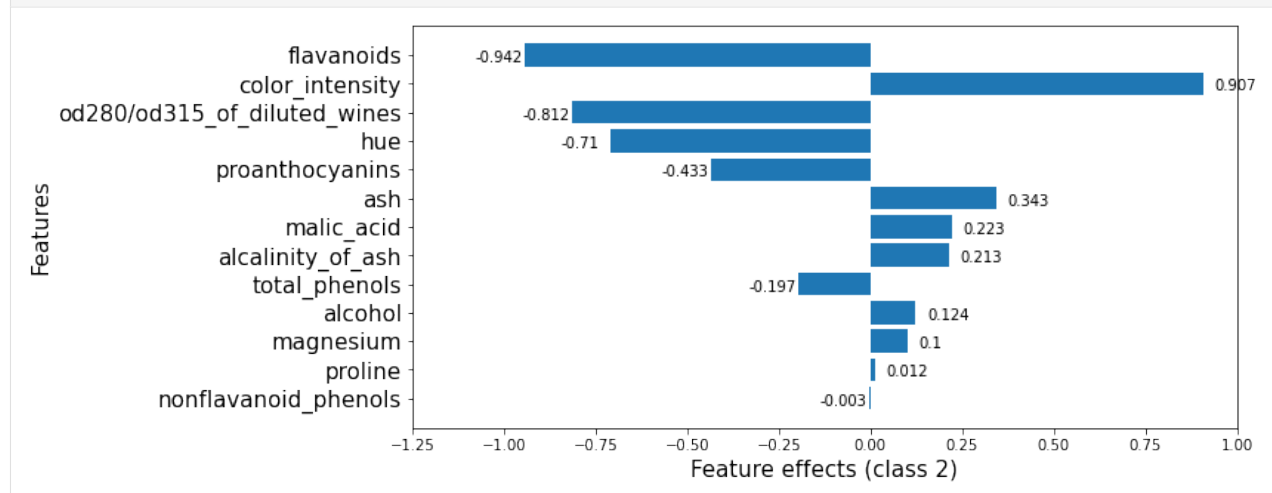
```
[16]: _, class_1_fig = plot_importance(feat_imp,
                                       feat_names,
                                       left_x=-1.5,
                                       right_x=1,
                                       eps_factor = 5, # controls text distance from end of
                                       ↪ bar for negative examples
                                       xlabel = "Feature effects (class {})".format(1),
                                       ylabel = "Features"
                                       )
```



```
[17]: feat_imp, feat_names = get_importance(2, # class_idx
      beta,
      feature_names,
      )
```

Finally, for class_2, the color_intensity, ash are the features that increase the class_2 odds.

```
[18]: _, class_2_fig = plot_importance(feat_imp,
      feat_names,
      left_x=-1.25,
      right_x=1,
      xlabel = "Feature effects (class {})".format(2),
      ylabel = "Features"
      # eps_factor = 5.
      )
```



Apply KernelSHAP to explain the model

Note that the *local accuracy* property of SHAP (eq. (5) in [1]) requires

$$f(x) = g(x') = \phi_0 + \sum_{j=1}^D \phi_j x'_j.$$

Hence, sum of the feature importances, ϕ_j , should be equal to the model output, $f(x)$. By passing `link='logit'` to the explainer, we ensure that ϕ_0 , the *base value* (see **Local explanation** section [here](#)) will be calculated in the correct units. Note that here $x' \in \mathbb{R}^D$ represents a *simplified input* for which the shap value is computed. A simple example of a simplified input in the image domain, justified by the dimensionality of the input space, is a *superpixel mask*: we formulate the task of explaining the outcome of an image prediction task as determining the effects of each superpixel in a segmented image upon the outcome. The interested reader is referred to [1] for more details about simplified inputs.

```
[19]: pred_fcn = classifier.predict_proba
lr_explainer = KernelShap(pred_fcn, link='logit')
lr_explainer.fit(X_train_norm)
```

Using 142 background data samples could cause slower run times. Consider using `shap.sample(data, K)` or `shap.kmeans(data, K)` to summarize the background as K samples.

```
[19]: KernelShap(meta={
    'name': 'KernelShap',
    'type': ['blackbox'],
    'task': 'classification',
    'explanations': ['local', 'global'],
    'params': {
        'link': 'logit',
        'group_names': None,
        'grouped': False,
        'groups': None,
        'weights': None,
        'summarise_background': False,
        'summarise_result': None,
        'transpose': False,
        'kwargs': {}
    },
    'version': '0.7.1dev'
})
```

```
[20]: # passing the logit link function to the explainer ensures the units are consistent ...
mean_scores_train = logit(pred_fcn(X_train_norm).mean(axis=0))
print(mean_scores_train - lr_explainer.expected_value)

[-3.33066907e-16  0.00000000e+00  3.33066907e-16]
```

```
[21]: lr_explanation = lr_explainer.explain(X_test_norm, l1_reg=False)

0%|          | 0/36 [00:00<?, ?it/s]
```

Because the dimensionality of the feature space is relatively small, we opted not to regularise the regression that computes the Shapley values. For more information about the regularisation options available for higher dimensional data see the introductory example [here](#).

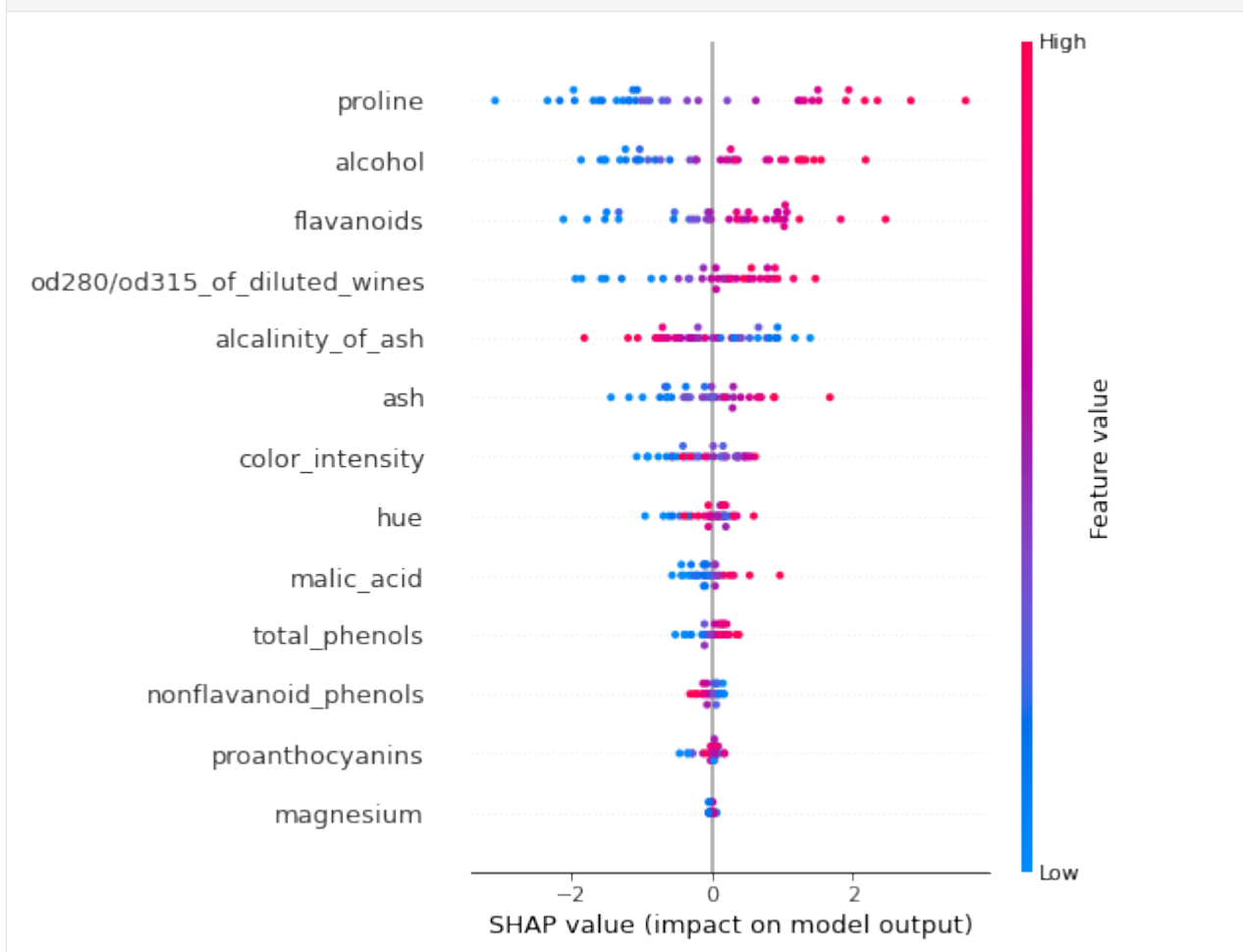
Locally explaining multi-output models with KernelShap

Explaining the logistic regression model globally with KernelSHAP

Summary plots

To visualise the impact of the features on the decision scores associated with class `class_idx`, we can use a summary plot. In this plot, the features are sorted by the sum of their SHAP values magnitudes across all instances in `X_test_norm`. Therefore, the features with the highest impact on the decision score for class `class_idx` are displayed at the top of the plot.

```
[22]: shap.summary_plot(lr_explanation.shap_values[class_idx], X_test_norm, feature_names)
```



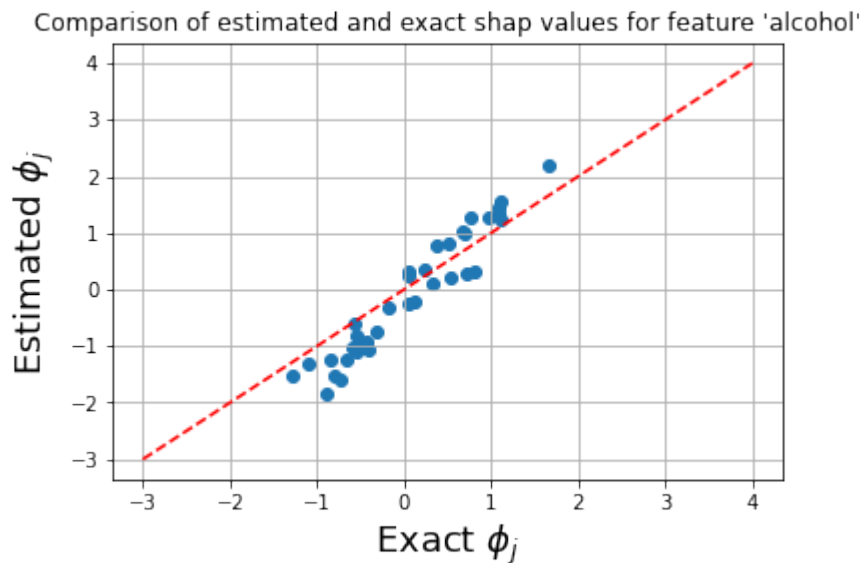
Because the logistic regression model uses a linear predictor function, the exact shap values for each class k can be computed exactly according to ([1])

$$\phi_{i,j}(f, x_i) = \beta_{j,k}(x_{i,j} - \mathbb{E}_{\mathcal{D}}[x_j]).$$

Here we introduced an additional index i to emphasize that we compute a shap value for *each predictor* and *each instance* in a set to be explained. This allows us to check the accuracy of the SHAP estimate. Note that we have already applied the normalisation so the expectation is not subtracted below.

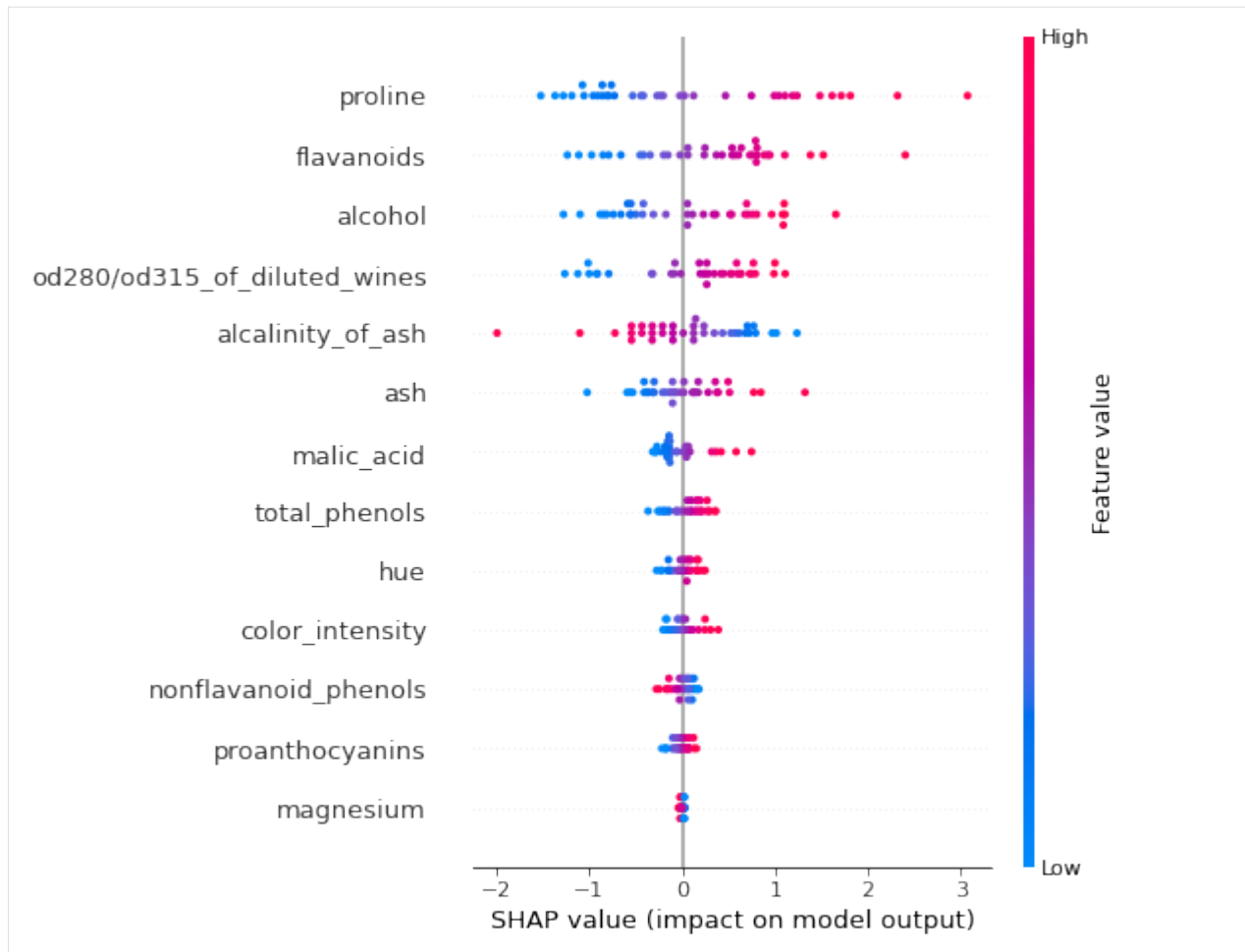
```
[23]: exact_shap = beta[:, None, :]*X_test_norm
```

```
[24]: feat_name = 'alcohol'
      feat_idx = feature_names.index(feat_name)
      x = np.linspace(-3, 4, 1000)
      plt.scatter(exact_shap[class_idx,...][:, feat_idx], lr_explanation.shap_values[class_
      → idx][:, feat_idx])
      plt.plot(x, x, linestyle='dashed', color='red')
      plt.xlabel(r'Exact  $\phi_j$ ', fontsize=18)
      plt.ylabel(r'Estimated  $\phi_j$ ', fontsize=18)
      plt.title(fr"Comparison of estimated and exact shap values for feature '{feat_name}'")
      plt.grid(True)
```



The plot below shows that the exact shap values and the estimate values give rise to similar ranking of the features, and only the order of the flavanoids and alcohol features is swapped.

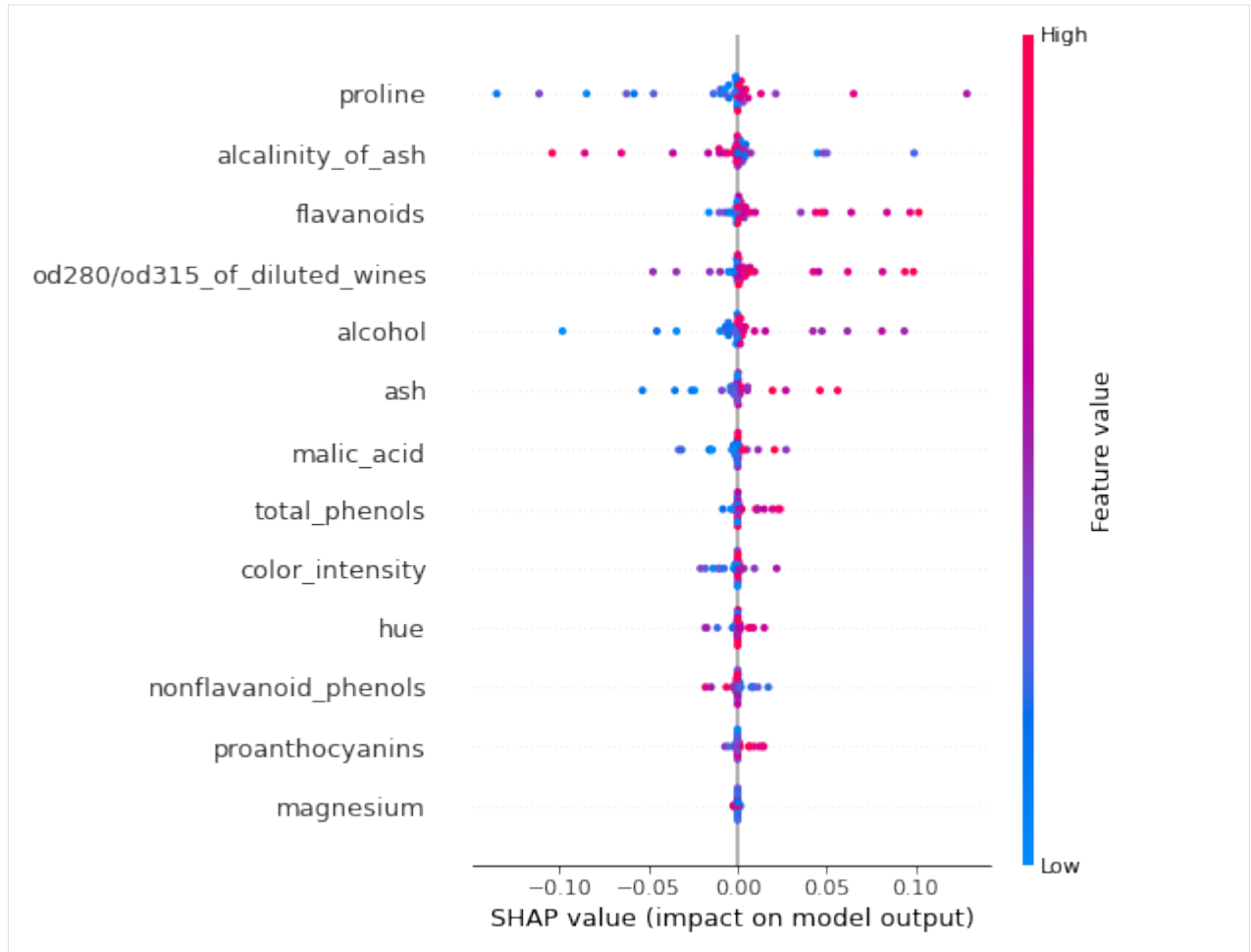
```
[25]: shap.summary_plot(exact_shap[class_idx, ...], X_test_norm, feature_names)
```



An similar plot can be create for the logistic regression model by plotting the marginal effects. Note that the plot labelling cannot be changed, so the x axis is incorrectly labeled as SHAP value below.

```
[26]: p = classifier.predict_proba(X_test_norm)
prb = p * (1. - p)
marg_effects = all_coefs[:, None, :] * prb.T[:, None]
assert (all_coefs[0, 0] * prb[:, 0] - marg_effects[0, :, 0]).sum() == 0.0
avg_marg_effects = np.mean(marg_effects, axis=1) # nb: ranking of the feature coefs
↳ should be preserved
mask = np.ones_like(X_test_norm) # the effect (postive vs negative) on the output
↳ depend on the sign of the input
mask[X_test_norm < 0] = -1

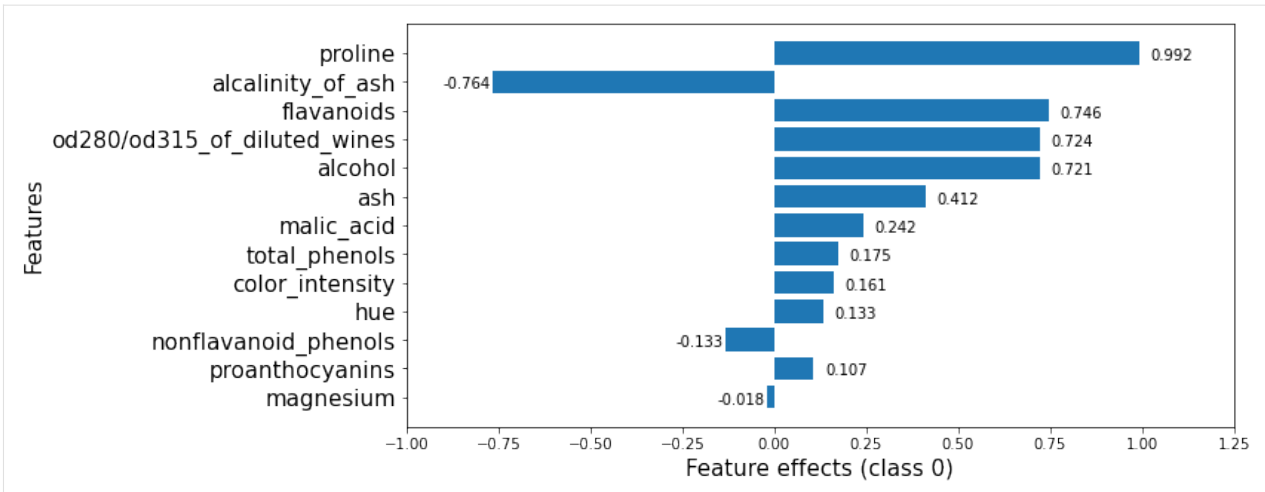
[27]: shap.summary_plot(marg_effects[class_idx, :, :-1]*mask, X_test_norm, feature_names) #
↳ exclude bias
```

As expected, the ranking of the marginal effects is the same as that provided the ranking the raw coefficients (see below). However, this effect measure allows us to assess the effects at instance level. Note that both the approximate computation and the exact method yield the same group of features as the most important, although their rankings are not identical. It is important to note that the exact effects ranking and absolute values is a function of the entire data (due to the dependence of the model coefficients) whereas the approximate computation is *local*: the explanation model is fitted locally around each instance. We also notice that the approximate and exact shap value computation both identify the same relationship between the feature value and the effect on the evidence of a sample belonging to `class_idx`.

```
[28]: class_0_fig
```

[28]:



Looking at the 6 most important features for this classification in `class_0`, we see that both the KernelSHAP method and the logistic regression rank the `proline` feature as the one with the most significant effect. While the order of the subsequent 5 features is permuted, the effects of these features are also very similar so, in effect, similar conclusions would be drawn from analysing either output.

References

- [1] Lundberg, S.M. and Lee, S.I., 2017. A unified approach to interpreting model predictions. In Advances in neural information processing systems (pp. 4765-4774).
- [2] Mood, C., 2017. “Logistic regression: Uncovering unobserved heterogeneity.”

8.10 Partial Dependence

8.10.1 Partial Dependence and Individual Conditional Expectation for predicting bike renting

In this example we will explain the behavior of a regression model on the [Bike rentals\[1\]](#) dataset. We will show how to calculate the partial dependence (PD) and the individual conditional expectation (ICE) to determine the feature effects on the model.

We will follow the example from the PDP chapter of the [Interpretable Machine Learning\[2\]](#) book and use the cleaned version of the dataset from the [github repository](#).

```
[1]: import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEncoder
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

from alibi.explainers import PartialDependence, plot_pd
```

Read and process the dataset

```
[2]: df = pd.read_csv('https://raw.githubusercontent.com/christophM/interpretable-ml-book/
↳ master/data/bike.csv')
df.head()
```

```
[2]:
```

	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	\
0	WINTER	2011	JAN	NO	HOLIDAY	SAT	NO WORKING DAY	MISTY	8.175849
1	WINTER	2011	JAN	NO	HOLIDAY	SUN	NO WORKING DAY	MISTY	9.083466
2	WINTER	2011	JAN	NO	HOLIDAY	MON	WORKING DAY	GOOD	1.229108
3	WINTER	2011	JAN	NO	HOLIDAY	TUE	WORKING DAY	GOOD	1.400000
4	WINTER	2011	JAN	NO	HOLIDAY	WED	WORKING DAY	GOOD	2.666979

	hum	windspeed	cnt	days_since_2011
0	80.5833	10.749882	985	0
1	69.6087	16.652113	801	1
2	43.7273	16.636703	1349	2
3	59.0435	10.739832	1562	3
4	43.6957	12.522300	1600	4

We will be using the `cnt` column as the target in the regression task. The `cnt` stands for the Count of bicycles which includes the casual and the registered users. We invite the reader to follow [this](#) link for more details on the dataset.

```
[3]: # extract feature names
feature_names = df.columns.tolist()
feature_names.remove('cnt')

# define target names
target_names = ['Number of bikes']

# define categorical columns
categorical_columns_names = ['season', 'yr', 'mnth', 'holiday', 'weekday', 'workingday',
↳ 'weathersit']

# define categorical and numerical indices for later preprocessing
categorical_columns_indices = [feature_names.index(cn) for cn in categorical_columns_
↳ names]
numerical_columns_indices = [feature_names.index(fn) for fn in feature_names if fn not_
↳ in categorical_columns_names]

# extract data
X = df[feature_names]
y = df['cnt']

# split data in train & test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

To preprocess the dataset in a format expected by the alibi explainers, we ordinally encode the categorical columns (i.e. string to integer) and we construct the `categorical_names` necessary to specify to the explainer which are the categorical features of the datasets. The `categorical_names` is a dictionary having as key the indices of the categorical columns and as values the corresponding feature values. For more details, see the [method description page](#).

```
[4]: # define and fit the ordinal encoder
oe = OrdinalEncoder().fit(X_train[categorical_columns_names])

# transform the categorical columns to ordinal encoding
X_train.loc[:, categorical_columns_names] = oe.transform(X_train[categorical_columns_
↳ names])
X_test.loc[:, categorical_columns_names] = oe.transform(X_test[categorical_columns_
↳ names])

# convert data to numpy
X_train, y_train = X_train.to_numpy(), y_train.to_numpy()
X_test, y_test = X_test.to_numpy(), y_test.to_numpy()

# define categorical mappings
categorical_names = {i: list(v) for (i, v) in zip(categorical_columns_indices, oe.
↳ categories_)}
```

We apply standard preprocessing steps to the dataset: standardization for the numerical features and one-hot encoding for the categorical ones. Note that the one-hot encoding is the representation to be used by the classifier. We require the previous label encoding step to transform the data into the standard format used by the alibi explainers.

```
[5]: # define numerical standard sclae
num_transf = StandardScaler()

# define categorical one-hot encoder
cat_transf = OneHotEncoder(
    categories=[range(len(x)) for x in categorical_names.values()],
    handle_unknown='ignore',
)

# define preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', cat_transf, categorical_columns_indices),
        ('num', num_transf, numerical_columns_indices),
    ],
    sparse_threshold=0
)
```

```
[6]: # fit preprocessor
preprocessor.fit(X_train)

# preprocess train and test datasets
X_train_ohe = preprocessor.transform(X_train)
X_test_ohe = preprocessor.transform(X_test)
```

Train regressor

Now that we have the dataset in a good format, we are ready to train the model. For this example, we use a `RandomForestRegressor` from `sklearn` library.

```
[7]: # define and fit regressor - feel free to play with the hyperparameters
predictor = RandomForestRegressor(random_state=0)
predictor.fit(X_train_ohe, y_train)

# compute scores
print('Train score: %.2f' % (predictor.score(X_train_ohe, y_train)))
print('Test score: %.2f' % (predictor.score(X_test_ohe, y_test)))

Train score: 0.98
Test score: 0.90
```

Partial dependence

Before proceeding with the explanation, there is one additional step we need to perform. The `PartialDependence` explainer expects the categorical features to be ordinal encoding and does not have explicit support for one-hot encoding yet (to be addressed in future releases).

To address this limitation, we can simply define a prediction function which applies the preprocessing step before passing the data to the `predict` method. This can be achieved as follows:

```
[8]: prediction_fn = lambda x: predictor.predict(preprocessor.transform(x))
```

Now that we defined the prediction function, we are ready to initialize the explainer.

```
[9]: # define explainer
explainer = PartialDependence(predictor=prediction_fn,
                             feature_names=feature_names,
                             target_names=target_names,
                             categorical_names=categorical_names)
```

Select a few features of interest, such as *temperature*, *humidity*, *wind speed*, and *season*.

```
[10]: # select temperature, humidity, wind speed, and season
features = [feature_names.index('temp'),
            feature_names.index('hum'),
            feature_names.index('windspeed'),
            feature_names.index('season')]
```

To compute the PD for the features listed we call the `explain` method. The parameter `kind='average'` specifies to return of the PD values. For some tree-based `sklearn` models, one can use the `TreePartialDependence` explainer for which the computation is faster. Note that the two methods do not agree in general on the values they return. This is because the marginal effect is computed with respect to different probability distributions. For more details on the computation method, check the [sklearn documentation page](#).

Following the PD computation, we can simply display the PD curves by calling the `plot_pd` method. The method allows the user to customize the plots as desired. For more details, see the [method description page](#).

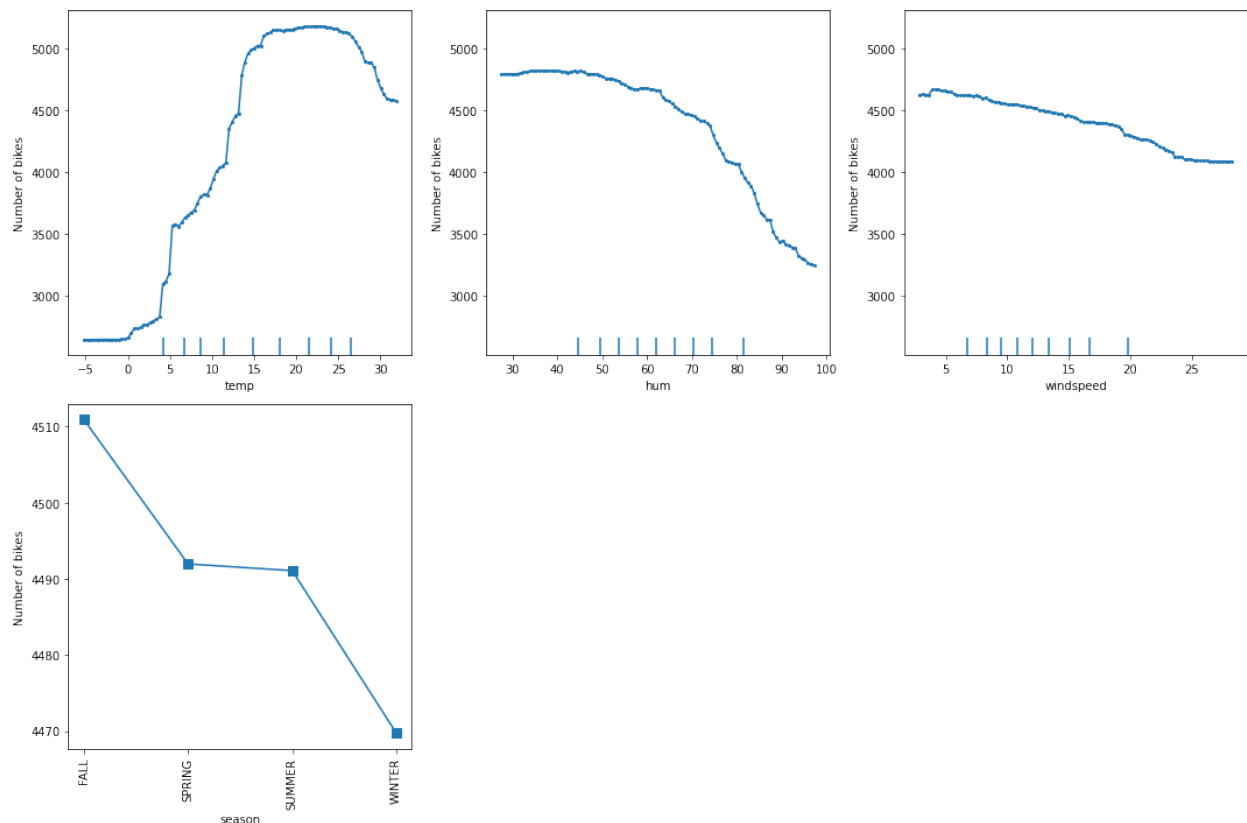
```
[11]: # compute explanations
exp = explainer.explain(X=X_train,
```

(continues on next page)

(continued from previous page)

```
features=features,
kind='average')
```

```
[12]: # plot partial dependence curves
plot_pd(exp=exp,
        n_cols=3,
        sharey='row',
        fig_kw={'figheight': 10, 'figwidth': 15});
```



We can observe that the average model prediction increases with the temperature till it reaches approximately 17°C . Then it flattens at a high number until the weather becomes too hot (i.e. approx. 27°C), after which it starts dropping again.

The humidity larger than 60% seems to be a factor that inhibits the number of rentals since we can observe a downward trend from that point onward.

A similar analysis can be conducted for the wind speed. As the wind speed increases, fewer and fewer people are riding the bike. Interestingly, as also mentioned in [here](#), the number of bike rentals flattens after 25km/h. By looking at the decile ticks, we can observe that there is not much data in that intervals. The model might not have learned to extrapolate correctly in that region, thus the predictions might not be meaningful.

Lastly, looking at the average prediction for each season, we can observe that all seasons show a similar effect on the model predictions, with a maximum in fall and a minimum in winter.

Individual conditional expectation

Although the PD plots can give us some insight concerning the average model response, they can also hide some heterogeneous effects. This is because the PD plots show the average marginal effects. To visualize the response of each data point and uncover heterogeneous effects we can use the ICE plots.

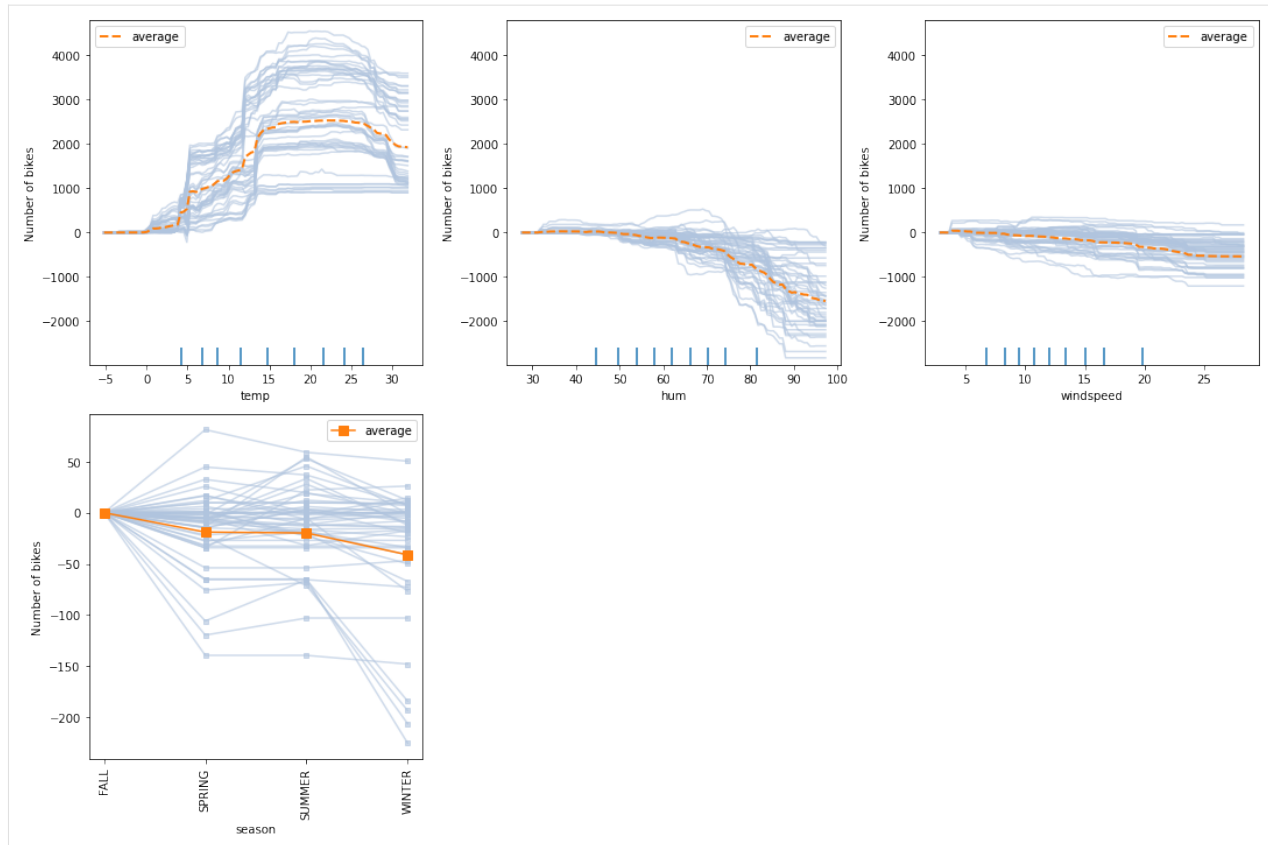
To compute both the PD and the ICE, we simply set the parameter `kind='both'`. Note that the `TreePartialDependence` alternative cannot compute the ICE.

Because the PD is the average of the ICE, the ICE plots can be heavily dispersed around the PD which can hide away the evolution of a data point as we change the feature value. Thus, it is recommended to center the plots at 0 by subtracting the value corresponding to the initial feature value.

```
[13]: # compute explanations
exp = explainer.explain(X=X_train,
                        features=features,
                        kind='both')

[14]: # random seed for `ice` sampling
np.random.seed(13)

# plot the pd and ice
plot_pd(exp=exp,
        n_cols=3,
        n_ice=50, # number of ICE curves to be displayed. Can be set to 'all' or
        ↪ provided a list of indices
        sharey='row',
        center=True, # center the plots for better visualization
        fig_kw={'figheight': 10, 'figwidth': 15});
```



For example, we can observe that there exist some particular scenarios in which the bike rental increases significantly and stays constant as the wind speed increases between 5 and 22km/h. Similarly, in some scenarios, the bike rental may stay the almost the same for winter relative to the other seasons. Such effects were hidden from us in the PD plot.

Partial dependence for two features

We will continue to provide some examples and a brief analysis of two feature PD plots, including a combination of two numerical features, two categorical features, and one numerical & one categorical. As we will see, 2-way PD plots helps us understand and visualize feature interactions.

```
[15]: features = [
        (feature_names.index('temp'), feature_names.index('windspeed')),
        (feature_names.index('mnth'), feature_names.index('weathersit')),
        (feature_names.index('season'), feature_names.index('temp'))
    ]
```

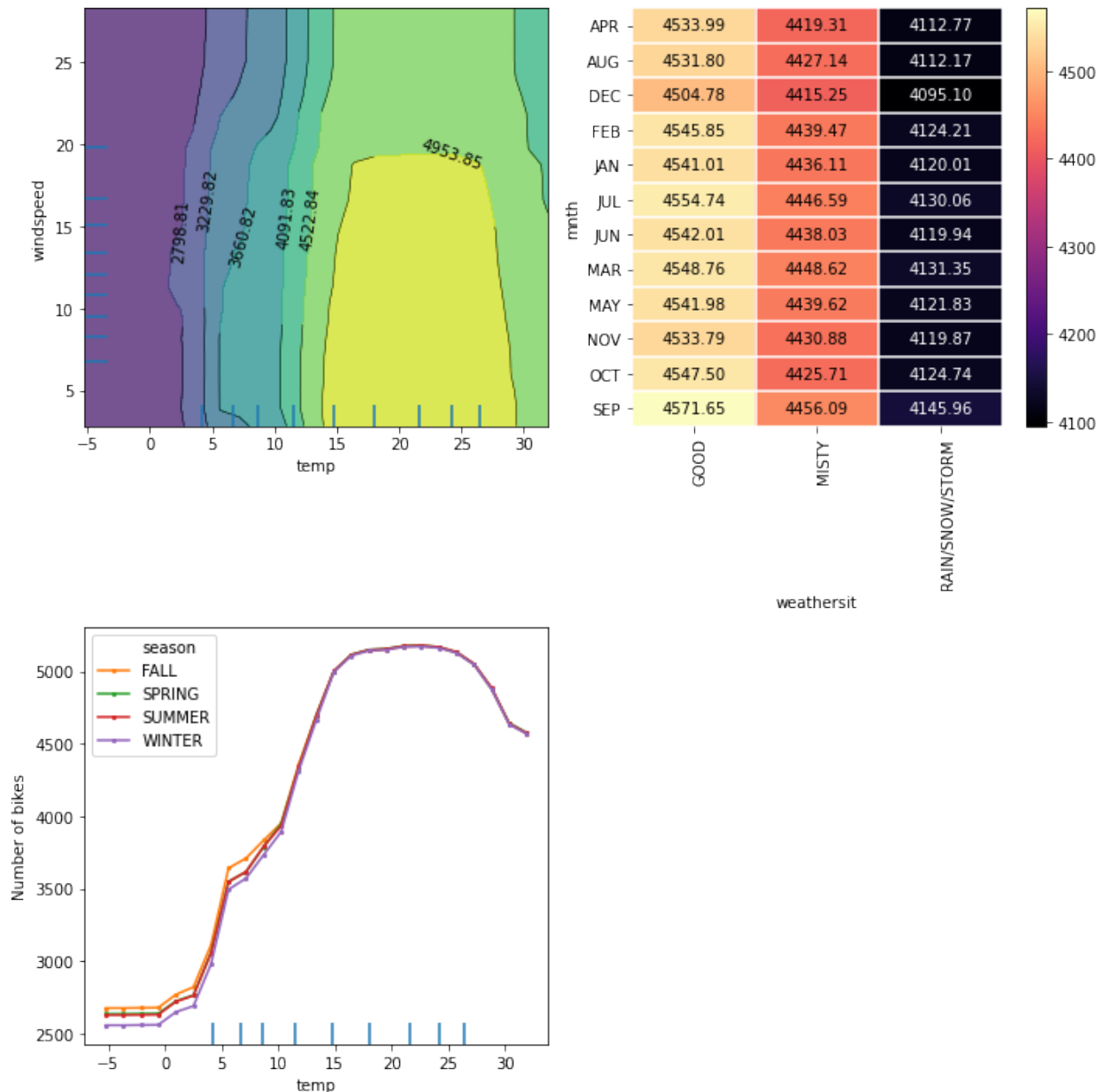
```
[16]: # compute explanations
exp = explainer.explain(X=X_train,
                        features=features,
                        kind='average',
                        grid_resolution=25)
```

```
[17]: # plot partial dependence curves
plot_pd(exp=exp,
```

(continues on next page)

(continued from previous page)

```
n_cols=2,
fig_kw={'figheight': 10, 'figwidth': 10});
```



From the interaction plot between the temperature and wind speed, we can observe that between -5 and about 12°C , the wind speed does not influence the average prediction that much. This can be deduced from the vertical strips of similar values restricted to the feature values domain of the reference dataset. The global trend is such that no matter the wind speed, the number of rented bikes increases with the temperature. As the temperature increases over 12°C and stays below 30°C , we can observe that the wind speed starts to become more relevant for the model's prediction. We note that the number of rented bikes stays high until the wind speed surpasses the value of approximately 18km/h , after which it starts dropping. This suggests that for relatively warm weather, the number of rentals increases as long as the wind is not too rough. Finally, we can observe that for extremely hot weather, the number of rentals drops again with the temperature.

By inspecting the weather situation against the month, interestingly, we can observe the rental prediction is influenced by the weather situation and not by the calendar month. As the weather deteriorates, the rentals drop, independent of the calendar month.

A similar situation can be observed in the plot of temperature against the season. Something that we've seen before is that the number of predicted rentals seems to be independent of the season and only depends on the temperature outside. As we have mentioned before, as the temperature increases, the number of rentals seems to increase till it reaches 17°C . Then it flattens at a high number until the weather becomes too hot (i.e. approx. 27°C), after which it starts dropping again.

References

- [1] Fanaee-T, Hadi, and Gama, Joao, 'Event labeling combining ensemble detectors and background knowledge', Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg.
- [2] Molnar, Christoph. Interpretable machine learning. Lulu. com, 2020.

8.11 Partial Dependence Variance

8.11.1 Feature importance and feature interaction based on partial dependence variance

In this notebook example we will explain the global behavior of a regression model trained on a synthetic dataset. We will show how to compute the global feature attribution and the feature interactions for a given model.

We will follow the example from [Greenwell et al. \(2018\)\[1\]](#) on the Friedman's regression problem defined in [Friedman et al. \(1991\)\[2\]](#) and [Breiman et al. \(1996\)\[3\]](#).

```
[1]: import numpy as np
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from alibi.explainers.pd_variance import PartialDependenceVariance, plot_pd_variance
```

Friedman's regression problem

Friedman's regression problem introduced in [Friedman et al.\[2\]](#) and [Breiman et al.\[3\]](#) consists in predicting a target variable based on ten independent features sample from a $\text{Uniform}(0, 1)$. Although the feature space consists of ten features, only the first five of them appear in the true model.

The relation between the input features and the response variables, y , is given by:

$$y = 10 \sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + \epsilon$$

where x_i , for $i = 1, \dots, 10$ are the ten input features, and $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

In the following cell, we generate a dataset of 1000 examples which we split into 500 training examples and 500 testing examples. Similar to the paper setup, the simulated observation are generated using a $\sigma = 1$.

```
[2]: def generate_target(X: np.ndarray):
    """
    Generates the target/response variable for the Friedman's regression problem.
```

(continues on next page)

(continued from previous page)

```

Parameters
-----
X
    A matrix realisations sample from a Uniform(0, 1). The size of the matrix is `N_
    ↪x 10`,
    where `N` is the number of data instances.

Returns
-----
Response variable.
"""
return 10 * np.sin(np.pi * X[:, 0] * X[:, 1]) + 20 * (X[:, 2] - 0.5)**2 \
    + 10 * X[:, 3] + 5 * X[:, 4] + np.random.randn(len(X))

np.random.seed(0)
X = np.random.rand(1000, 10)
y = generate_target(X)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=0)

```

Train MLP regressor

Similar with [Greenwell et al.\[1\]](#), we train a Muti-layer Perceptron (MLP) regressor and report its score on both the train and test split. For the purposes of this examples, we keep the default configuration for the MLPRegressor.

```

[3]: # train MLP regressor on data
nn = MLPRegressor(max_iter=10000, random_state=0)
nn = nn.fit(X_train, y_train)

# compute score on train and test dataset
print(f"Train score: {nn.score(X_train, y_train):.3f}")
print(f"Test score: {nn.score(X_test, y_test):.3f}")

Train score: 0.968
Test score: 0.931

```

Define explainer

Now that we have the prediction model, we can define the PartialDependenceVariance explainer to compute the feature importance and feature interactions.

Note that our explainer can work with any black-box model by providing the prediction function, which in our case will be `nn.predict`. Furthermore, we can specify the feature names and the target names to match our formulation through the parameters `feature_names` and `target_names`.

```

[4]: # define explainer
explainer = PartialDependenceVariance(predictor=nn.predict,
                                     feature_names=[f'x{i}' for i in range(1, 11)],
                                     target_names=['y'],
                                     verbose=True)

```

Feature importance

With our explainer initialized, we can compute the feature importance for all features through a simple call to the `explain` function. The arguments provided would be a reference dataset `X` which is usually the training dataset (i.e., `X_train` in our example) and setting `method='importance'`. Note that the `explain` function can receive many other arguments through which the user can specify explicitly the features to compute the feature importance for, the grid points (i.e., in our case `grid_resolution=50` to speed up the computation), etc. We refer the reader to our documentation page for further details.

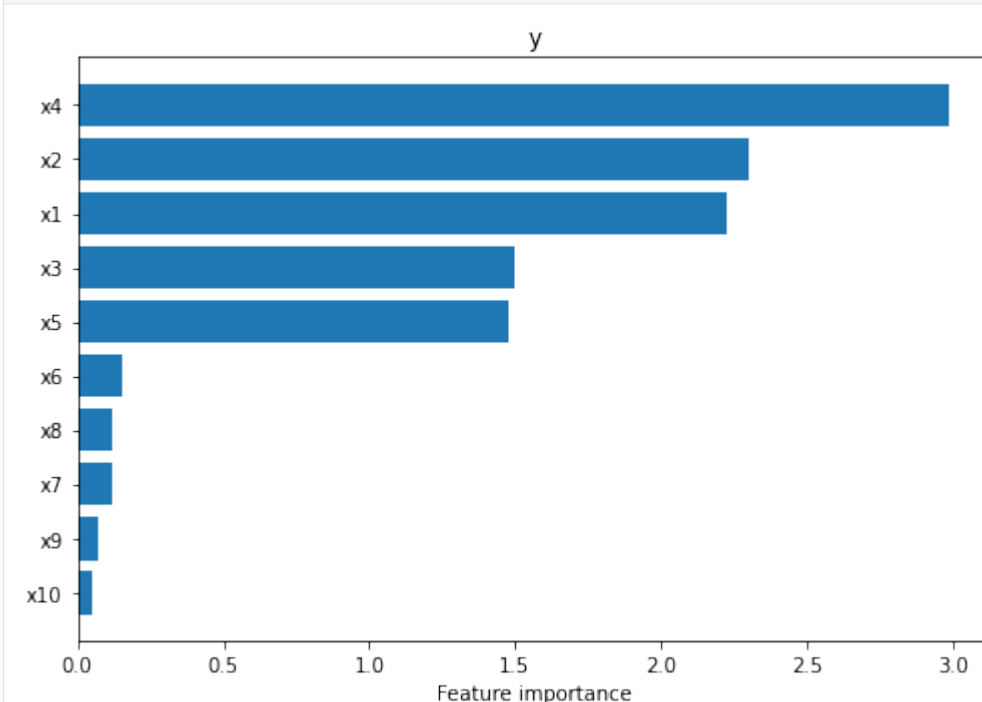
```
[5]: exp_importance = explainer.explain(X=X_train,
                                     method='importance',
                                     grid_resolution=50)
```

```
100%|=====| 10/10 [00:00<00:00, 28.22it/s]
```

Once our explanation is computed, we can visualize the feature importance in two ways. Alibi implements an utility plotting function, `plot_pd_variance`, which helps the user quickly visualize the results.

The simplest way is to visualize the results through a horizontal bar plot. By default, the features are ordered in descending order by their importance from top to bottom. This can be achieved through as simple call to `plot_pd_variance` as follows:

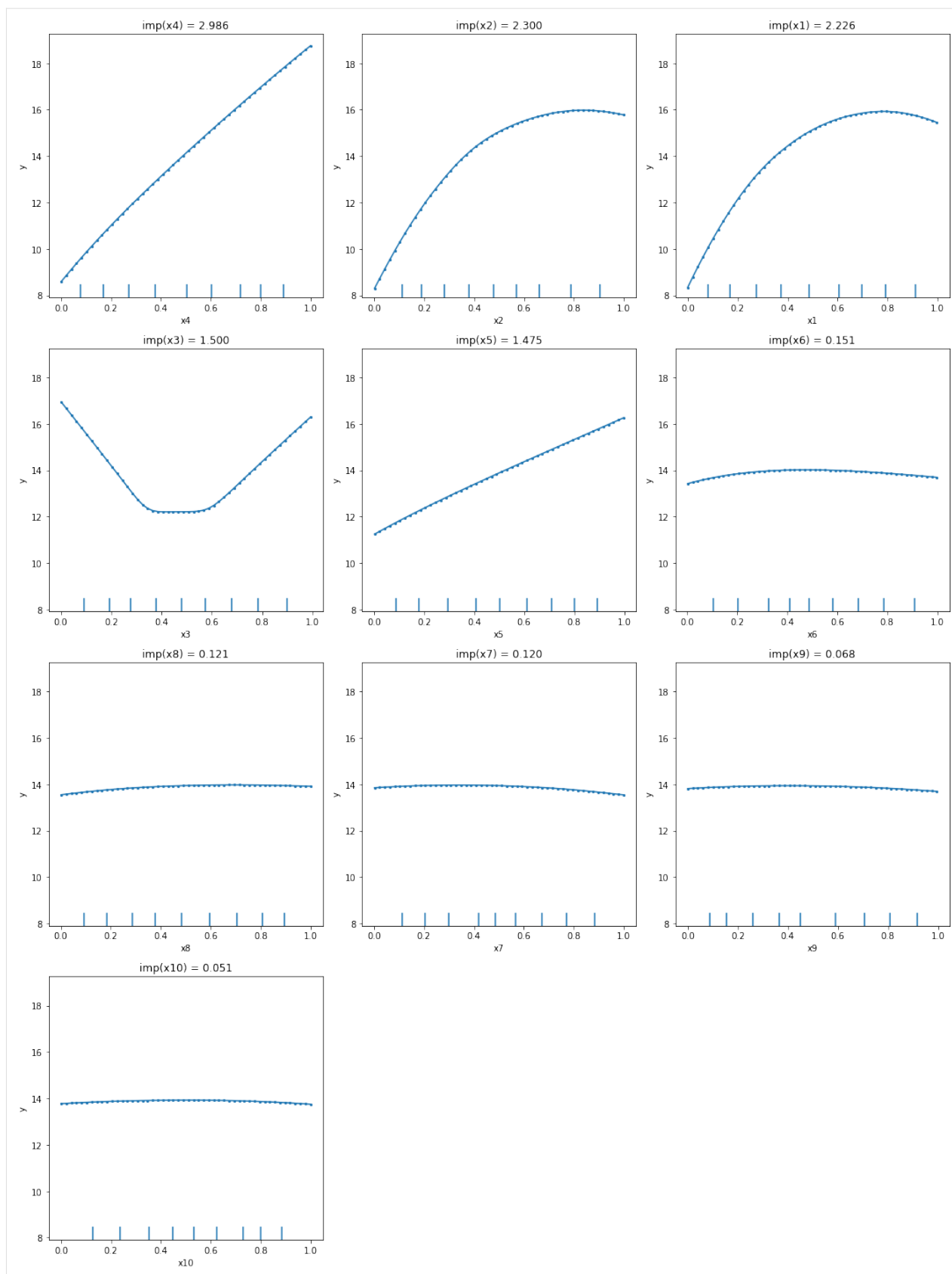
```
[6]: plot_pd_variance(exp=exp_importance,
                     features='all',
                     fig_kw={'figwidth': 7, 'figheight': 5});
```



We can see straight away that the explainer managed to identify that the first five features are the most salient (i.e., x_4, x_2, x_1, x_3, x_5 in decreasing order of their importance).

As also recommended in the paper, the feature importance should be analyzed concomitantly with the Partial Dependence Plots (PDP) described in [Friedman et al. \(2001\)\[4\]](#) and [Molnar \(2020\)\[5\]](#) based on which the importance has been calculated. Our utility function allows the user to visualize the PDPs by simply setting the parameter `summarise=False`. As before, the plots are sorted in descending order based on the corresponding feature importance.

```
[7]: plot_pd_variance(exp=exp_importance,
                      features='all',
                      summarise=False,
                      n_cols=3,
                      fig_kw={'figwidth': 15, 'figheight': 20});
```



From the PDPs, we can observe that the explainer managed to identify correctly the effects of each feature: linear for x_4 and x_5 , quadratic for x_3 , and sinusoidal for x_1 and x_2 . The other variables show a relative flat main effect which according to the method's assumption means a low importance. Also, by inspecting the plots we can see that x_4 main effect spans a range from 8 to somewhere around 19, which is probably one of the reasons why it got the largest importance.

Feature interaction

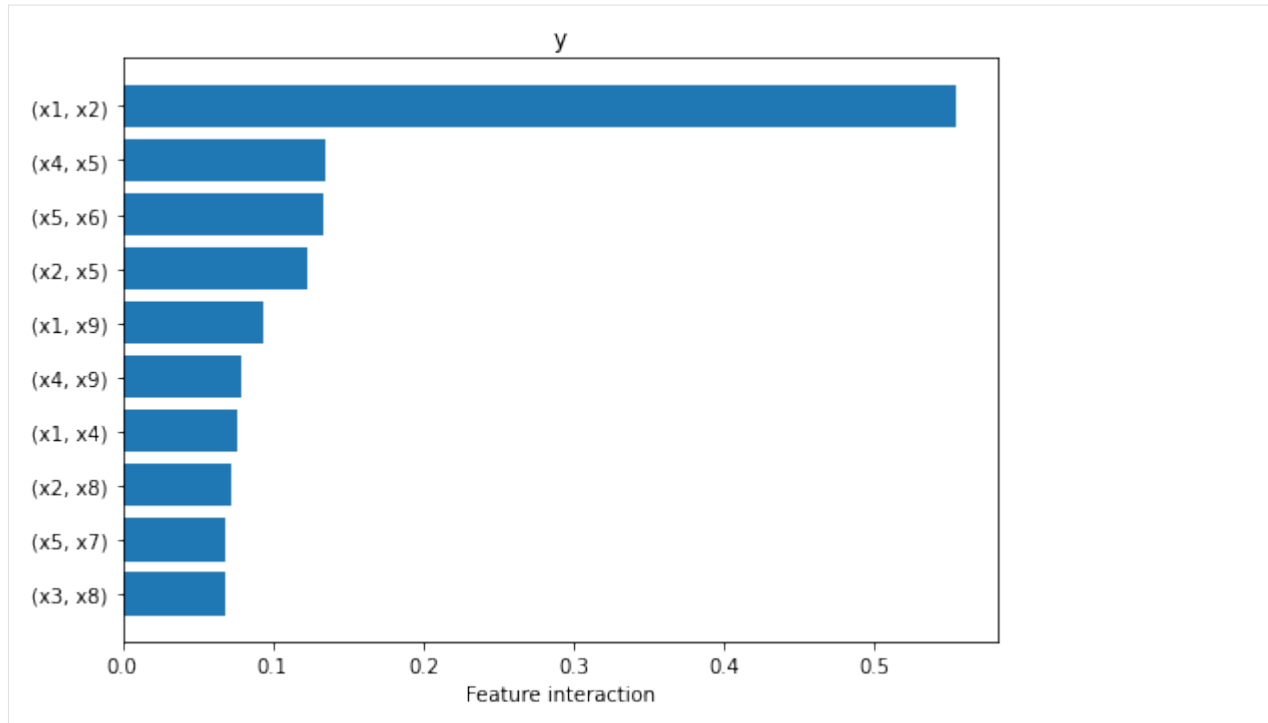
As previously mentioned, the `PartialDependenceVariance` explainer is able to compute a measure of feature interaction. The call to the explainer follows the same API as above, just by simply calling the `explain` function with the parameter `method='interaction'`. By default, the explainer will compute a measure of interaction for all possible pairs of features. Note that this is quadratic in the number of features and is based on computing a two-ways partial dependence function for all pairs. Thus, this step might be more computationally demanding. Similar with the computation of the feature importance, the user has the liberty to provide the features pairs for which the feature interaction will be computed and control the computation complexity through the grid parameters.

```
[8]: exp_interaction = explainer.explain(X=X_train,
                                     method='interaction',
                                     grid_resolution=30)
```

```
100%|=====| 45/45 [00:34<00:00, 1.32it/s]
```

Once the explanation is computed, we can visualize the summary horizontal plot to identify the pairs of features that interact the most. Because the plot can grow very tall due to the quadratic number of feature pairs, we expose the `top_k` parameter to limit the plot to the `top_k` most important features provided through the `features` parameter. In our case we set `top_k=10` and since `features='all'`, the plot will display the 10 feature pairs that interact the most out of all feature pairs.

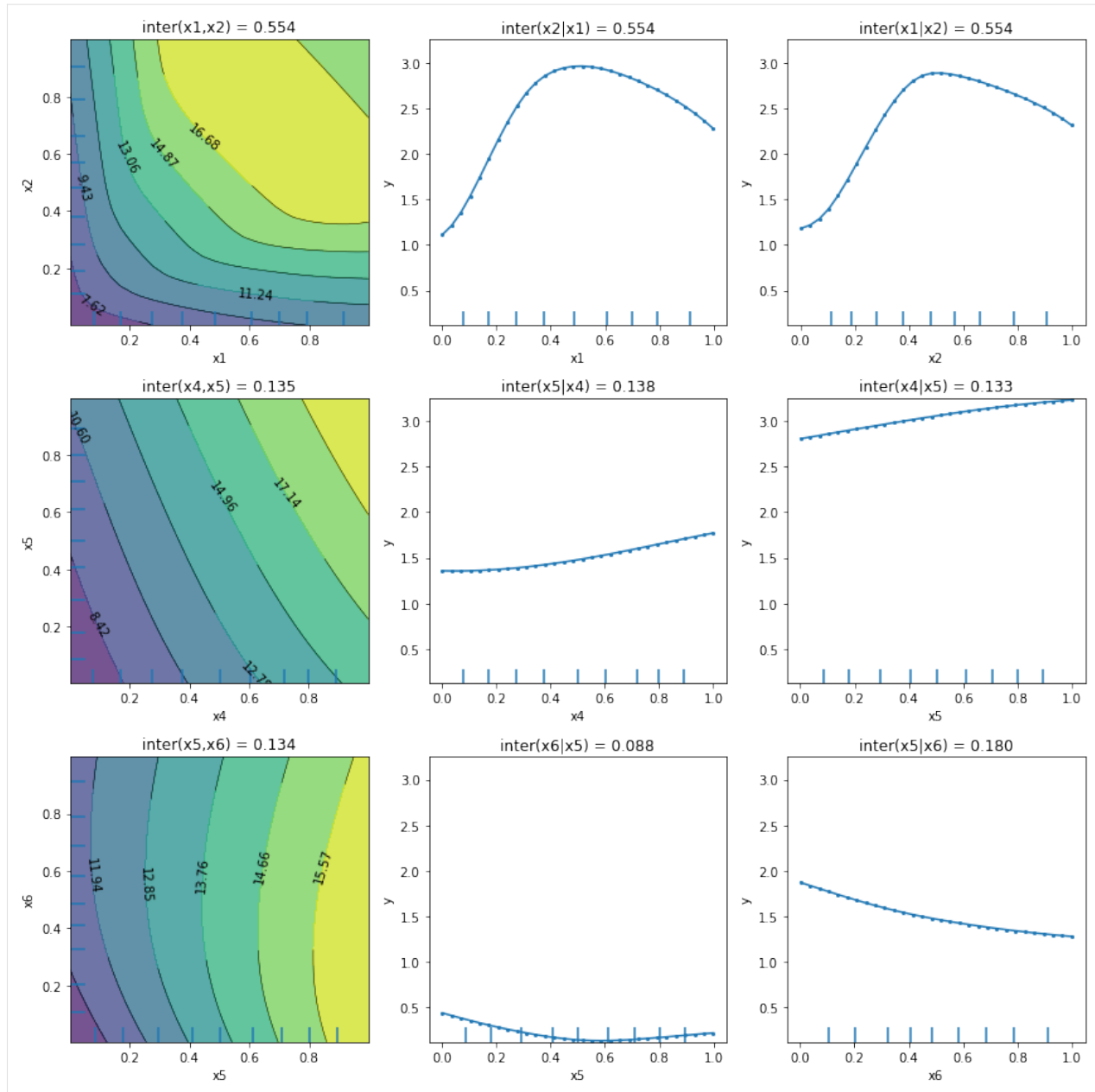
```
[9]: # plot summary
plot_pd_variance(exp=exp_interaction,
                 features='all',      # considers plotting all features
                 top_k=10,           # plots only the top 10 features from all the_
                 ↪ `features`
                 fig_kw={'figwidth': 7, 'figheight': 5});
```



From the plot above we can observe that the explainer attributes non-zero interaction values to many pairs of features, but interaction between x_1 and x_2 is the one that stands out, being an order of magnitude higher than the rest. This is in fact the only pair of features that interact through the function $\sin(\pi x_1 x_2)$.

As before, if we would like to visualize more details, we can call again the `plot_pd_variance` with `summarise=False`. For each explained feature pair the function will plot the two-way PDP followed immediately by two conditional importance plots, when conditioning on one feature at a time. Note that the final interaction between the two features is computed as the average of the two conditional interactions (see titles of each subplot). For visualization purposes, we recommend using `n_cols=3`, such that each row will describe only the feature interaction between one pair. Similar as before, the plots are displayed in descending order based on their interaction.

```
[10]: plot_pd_variance(exp=exp_interaction,
                      features='all', # considers plotting all feature pairs
                      top_k=3,       # plots only the top 3 features pairs from all the
                      ↪ `features`
                      summarise=False,
                      n_cols=3,
                      fig_kw={'figwidth': 12, 'figheight': 12});
```

We can observe that apart from the first plot corresponding to features x_1 and x_2 , the other plots present an almost flat trend which can be an indication, but not a guarantee, of the absence of feature interaction. There exist functions for which the `PartialDependenceVariance` explainer does not capture the feature interaction. We refer the reader to the [method description](#) for a more detailed example.

References

- [1] Greenwell, Brandon M., Bradley C. Boehmke, and Andrew J. McCarthy. “A simple and effective model-based variable importance measure.” arXiv preprint arXiv:1805.04755 (2018).
- [2] Friedman, Jerome H. “Multivariate adaptive regression splines.” The annals of statistics 19.1 (1991): 1-67.
- [3] Breiman, Leo. “Bagging predictors.” Machine learning 24.2 (1996): 123-140.
- [4] Friedman, Jerome H. “Greedy function approximation: a gradient boosting machine.” Annals of statistics (2001): 1189-1232.
- [5] Molnar, Christoph. Interpretable machine learning. Lulu. com, 2020.

8.12 Permutation Importance

8.12.1 Permutation Feature Importance on “Who’s Going to Leave Next?”

In this notebook example, we will explain the global behavior of a classification model by identifying the most important features it relies on. To obtain the importance of the features, we will use the `PermutationImportance` explainer, initially proposed by [Breiman \(2001\)\[1\]](#), and further refined by [Fisher et al. \(2019\)\[2\]](#).

This notebook is inspired from the following [blogpost](#).

Since `seaborn` is not a required dependency, we need to install it:

```
[ ]: !pip install -q seaborn
```

```
[1]: import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, f1_score, roc_auc_score

from alibi.explainers import PermutationImportance, plot_permutation_importance
```

Read the dataset

```
[2]: df = pd.read_csv('https://raw.githubusercontent.com/ucg8j/kaggle_HR/master/HR_comma_sep.
    ↪ csv')
df.head()
```

```
[2]:
```

	satisfaction_level	last_evaluation	number_project	average_monthly_hours	\
0	0.38	0.53	2	157	
1	0.80	0.86	5	262	
2	0.11	0.88	7	272	
3	0.72	0.87	5	223	

(continues on next page)

(continued from previous page)

	0.37	0.52	2	159
time_spend_company	Work_accident	left	promotion_last_5years	sales \
0	3	0	1	0 sales
1	6	0	1	0 sales
2	4	0	1	0 sales
3	5	0	1	0 sales
4	3	0	1	0 sales

	salary
0	low
1	medium
2	medium
3	low
4	low

We will be using the `left` column as the target for the binary classification task. A value of 1 in the `left` column indicates that a person left the company.

```
[3]: # define target column
target_name = 'left'

# extract the features
feature_names = df.columns.to_list()
feature_names.remove(target_name)

# define categorical columns
categorical_names = [
    'Work_accident',      # binary
    'promotion_last_5years', # binary
    'sales',              # nominal
    'salary'              # ordinal, but will treat it as nominal
]

# define numerical features
numerical_names = [ft for ft in feature_names if ft not in categorical_names]
```

Note that although the `salary` feature is ordinal, we will treat it as a nominal feature in this example.

Data analysis

Before diving into the data preprocessing step and the actual model training, let us first explore the dataset. We begin by inspecting the proportion of positive and negative instances available in our dataset.

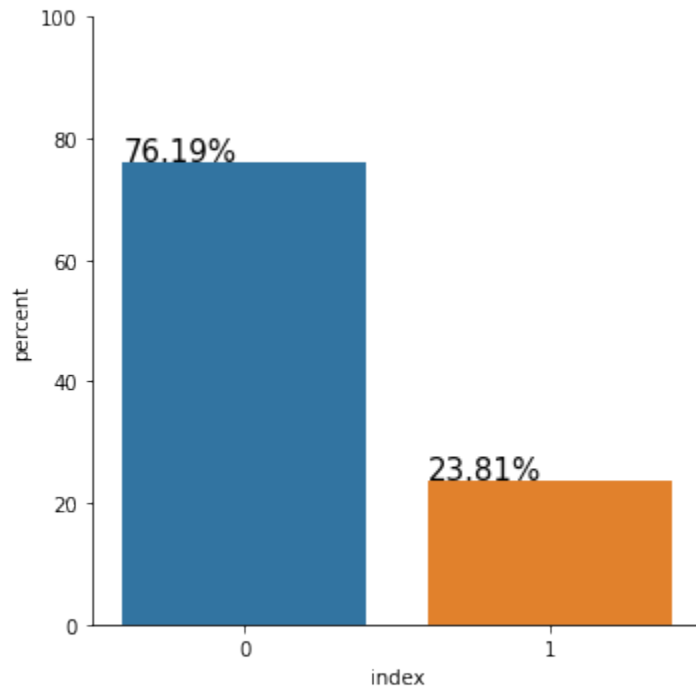
```
[4]: target_perc = df[target_name].value_counts(normalize=True).mul(100).rename('percent').
    ↪ reset_index().rename({'index': 'left'}, axis=1) # rename "index" on pandas 1.x
g = sns.catplot(data=target_perc, x='left', y='percent', kind='bar')
g.ax.set_ylim(0, 100)

for p in g.ax.patches:
    txt = str(p.get_height().round(2)) + '%'
```

(continues on next page)

(continued from previous page)

```
txt_x, txt_y = p.get_x(), p.get_height()
g.ax.text(x=txt_x, y=txt_y, s=txt, fontdict={'size': 15})
```



Right away, we can observe that our dataset is quite imbalanced. The people who left the company are the minority class, representing only 23.81% of the entire dataset. Although this might not be the case for us, with extreme class imbalance, we should carefully consider which metric to use to evaluate the performance of our model.

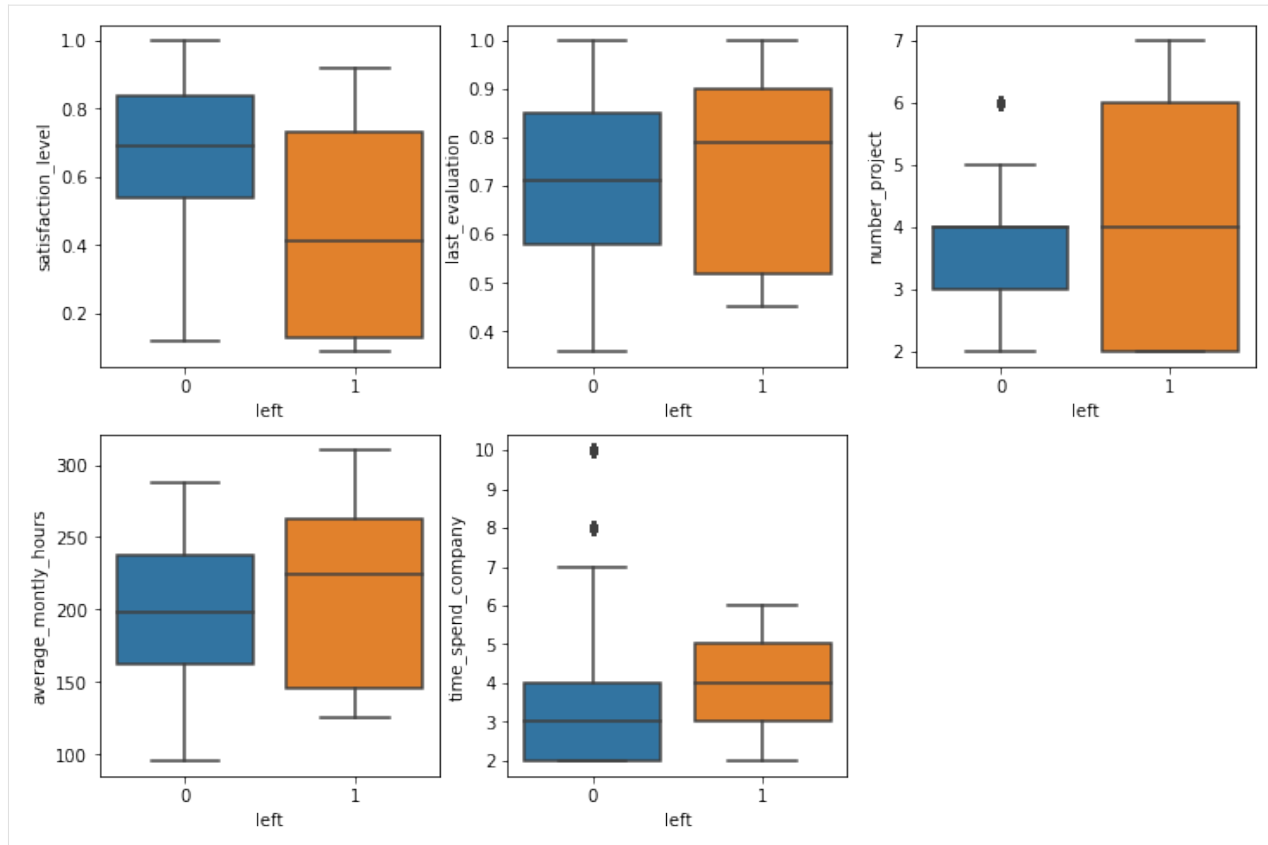
For example, reporting the accuracy alone might not be sufficient. Consider, for example, the case in which we have 99% of the data instances belonging to the negative class and 1% belonging to the positive class - the class of interest. A classifier which predicts 0 every time will achieve an accuracy of 99%. Although the model achieves a high accuracy it is useless because it cannot detect any positive instances. Thus, in this scenario, better metrics to inspect the model performance are the precision, the recall, and the F_1 score. In addition, one can analyze the ROC-AUC curve to measure class separation.

Let us now inspect the relationship between each feature and the target. We begin by looking at the distribution of the numerical features grouped by labels.

```
[5]: fig, axs = plt.subplots(nrows=2, ncols=3, figsize=(12, 8))
     axs = axs.flatten()

     for ax, ft in zip(axs, numerical_names):
         sns.boxplot(data=df, y=ft, x=target_name, ax=ax, orient='v')

     fig.delaxes(axs[-1])
```



By inspecting the distributions above, we already see some associations. It is probably not surprising that people with lower satisfaction levels are more likely to leave. Similarly, people who work more hours and people that are older in the company have the same tendency to leave. All those associations make intuitive sense, and one can propose multiple plausible hypotheses on why this happens.

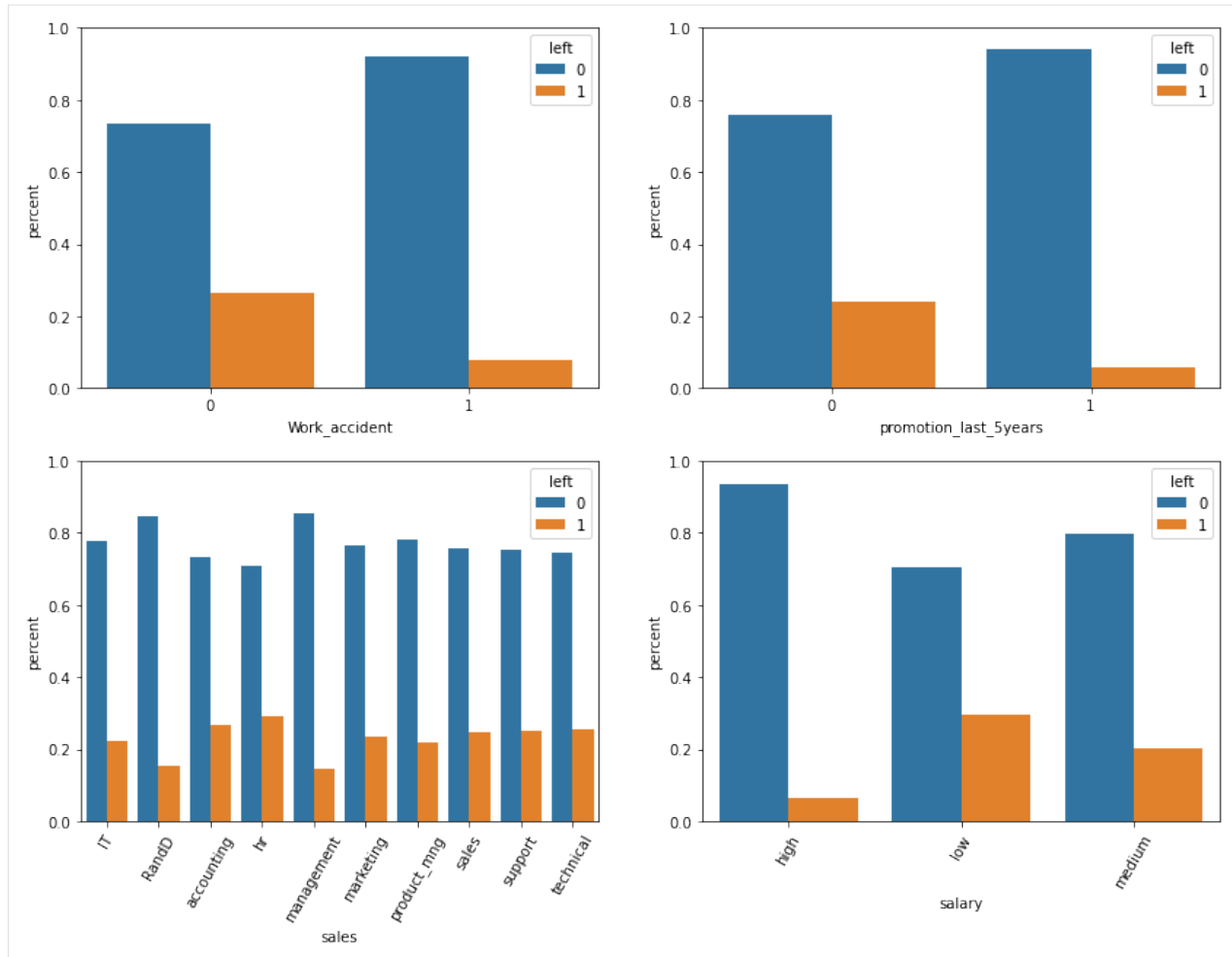
Quite interesting is that people who tend to have a higher evaluation score are also associated with leaving the company. Later, we will inspect the interactions between the two features, and we may be able to understand why this happens.

We now look at the categorical features.

```
[6]: fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(14, 10))
     axs = axs.flatten()

     for ax, ft in zip(axs, categorical_names):
         ser1 = df.groupby([target_name, ft])[target_name].count()
         ser2 = df.groupby([ft])[target_name].count()
         df_prec = (ser1 / ser2).rename('percent').reset_index()
         sns.barplot(data=df_prec, x=ft, y='percent', hue=target_name, ax=ax)
         ax.set_ylim(0, 1)

     axs[-2].tick_params(axis='x', rotation=60)
     axs[-1].tick_params(axis='x', rotation=60)
```



We can eyeball that some conditional distributions differ from the unconditional label distribution (76.19%, 23.81%):

```
[7]: prec = (df[df.Work_accident == 1].left == 0).mean() * 100
print("* Percentage staying if a work accident happended: {:.2f}% > 76.19%\n".
      ↪format(prec))

prec = (df[df.promotion_last_5years == 1].left == 0).mean() * 100
print("* Percentage staying if a promotion happend in the last 5 years: {:.2f}% > 76.19%\n".
      ↪format(prec))

prec = (df[df.sales == 'RandD'].left == 0).mean() * 100
print("* Percentage staying if the `sales` feature equals 'RandD': {:.2f}% > 76.19%\n".
      ↪format(prec))

prec = (df[df.sales == 'management'].left == 0).mean() * 100
print("* Percentage staying if the `sales` feature equals 'management': {:.2f}% > 76.19%\n".
      ↪format(prec))

prec = (df[df.salary == 'high'].left == 0).mean() * 100
print("* Percentage staying if they have a high salary: {:.2f}% > 76.19%\n".format(prec))
```

(continues on next page)

(continued from previous page)

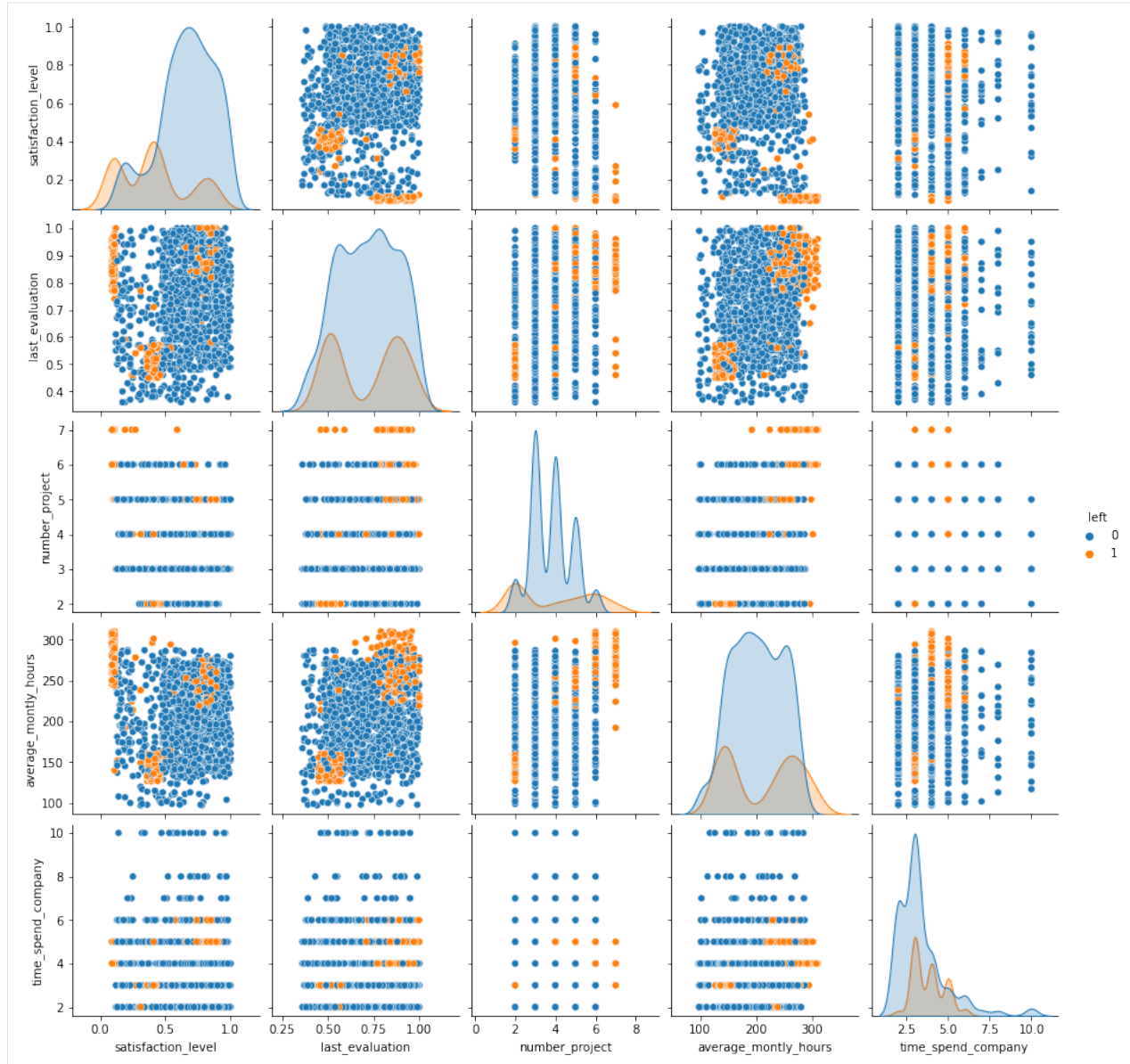
```
prec = (df[df.sales == 'hr'].left == 1).mean() * 100
print("* Percentage leaving if the feature `sales` equals 'hr': {:.2f}% > 23.81%".
      ↪format(prec))
```

```
* Percentage staying if a work accident happended: 92.21% > 76.19%
* Percentage staying if a promotion happend in the last 5 years: 94.04% > 76.19%
* Percentage staying if the `sales` feature equals 'RandD': 84.63% > 76.19%
* Percentage staying if the `sales` feature equals 'management': 85.56% > 76.19%
* Percentage staying if they have a high salary: 93.37% > 76.19%
* Percentage leaving if the feature `sales` equals 'hr': 29.09% > 23.81%
```

We can observe from the above computation that people are more likely to stay if they had a work accident, or they had a promotion in the last five years, or the sales feature equals 'RandD' or 'management', or of they have a high salary. Also, probably not as significant, we can observe a tendency to leave for people having the feature sales equal 'hr'.

We continue our analysis by visualizing the interactions between two numerical features and their association with the target label.

```
[8]: sns.pairplot(data=df.sample(frac=0.1), x_vars=numerical_names, y_vars=numerical_names,
      ↪hue=target_name);
```



There are quite a few interactions that are worth mentioning:

- (`satisfaction_level`, `last_evaluation`) - We can observe three groups that tend to leave the company. We have people with a satisfaction level around 0.4 and the last evaluation around 0.5. Those are people who are not very happy with their job and are not that great at their tasks, and thus it makes intuitive sense to leave the company. We also have people with low satisfaction levels and high evaluation scores. Those might be very skillful people who can easily find other opportunities when they are not pleased anymore with their job. Finally, we have the satisfied people with high-performance evaluations representing the ones who enjoy and are very good at what they are doing but probably leave for better opportunities.
- (`satisfaction_level`, `average_monthly_hours`) - Analogous to the previous case, we identify three groups (might coincide). The first group consists of people with a satisfaction level around 0.4 and who work on average a low number of hours per month. In the second group, we have people who work a lot, but are not very happy with their job. Finally, the third group is represented by satisfied people who work a lot.
- (`last_evaluation`, `number_project`) - we distinguish two groups. The first group consists of people with a low evaluation score and a low number of projects which might not be very productive for the company. The

second group consists of people with a high evaluation score and a high number of projects.

- (last_evaluation, average_monthly_hours) - Similarly, we can see two clear clusters defined by people with a low evaluation score and who work on average a low number of hours per month and people with a high evaluation score and large number monthly working hours.

Although there are many other interactions to mention, we stop here for the sake of this example. One can conduct similar investigations the categorical features. From the above analysis, we can conclude that numerical features are very relevant for the classification task.

Data preprocessing

We first split the dataset into train and test.

```
[9]: X = df[feature_names].to_numpy()
     y = df[target_name].to_numpy()

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,
     ↪random_state=0)
```

Define and fit the preprocessor. We use only one hot encoding (OHE) for the categorical variables.

```
[10]: categorical_indices = [feature_names.index(ft) for ft in categorical_names]

     # define categorical one-hot encoder
     cat_transf = OneHotEncoder(
         categories=[np.unique(X_train[:, ft_idx]) for ft_idx in categorical_indices],
         drop='if_binary'
     )

     # define preprocessor
     preprocessor = ColumnTransformer(
         transformers=[
             ('cat', cat_transf, categorical_indices)
         ],
         remainder='passthrough',
         sparse_threshold=0,
     )

     # fit the preprocessor
     preprocessor = preprocessor.fit(X_train)
```

With the preprocessor fitted, we compute the OHE representation of the training and testing dataset.

```
[11]: # get OHE data representation
     X_train_ohe = preprocessor.transform(X_train)
     X_test_ohe = preprocessor.transform(X_test)
```

Train and evaluate random forest classifier

Now that we have the dataset in a good format, we are ready to train the `RandomForestClassifier` from the `sklearn` library

```
[12]: rf = RandomForestClassifier(class_weight='balanced', random_state=0)
      rf = rf.fit(X_train_ohe, y_train)
```

```
[13]: # evaluate classifier on train data
      y_train_hat = rf.predict(X_train_ohe)
      print(classification_report(y_true=y_train, y_pred=y_train_hat))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	9142
1	1.00	1.00	1.00	2857
accuracy			1.00	11999
macro avg	1.00	1.00	1.00	11999
weighted avg	1.00	1.00	1.00	11999

```
[14]: # evaluate classifier on test data
      y_test_hat = rf.predict(X_test_ohe)
      print(classification_report(y_true=y_test, y_pred=y_test_hat))
```

	precision	recall	f1-score	support
0	0.99	1.00	0.99	2286
1	1.00	0.96	0.98	714
accuracy			0.99	3000
macro avg	0.99	0.98	0.99	3000
weighted avg	0.99	0.99	0.99	3000

As we can observe, our classifier performs reasonably well on all the metrics of interest.

Permutation importance

With our classifier trained, we can perform post-hoc explanation to determine which features are the most important for our model. We begin by defining a prediction function followed by the initialization of the alibi explainer. Note that alibi supports some metric functions which can be specified through strings.

```
[15]: def predict_fn(X: np.ndarray) -> np.ndarray:
      return rf.predict(preprocessor.transform(X))
```

```
[16]: explainer = PermutationImportance(predictor=predict_fn,
      score_fns=['accuracy', 'f1'],
      feature_names=feature_names,
      verbose=True)
```

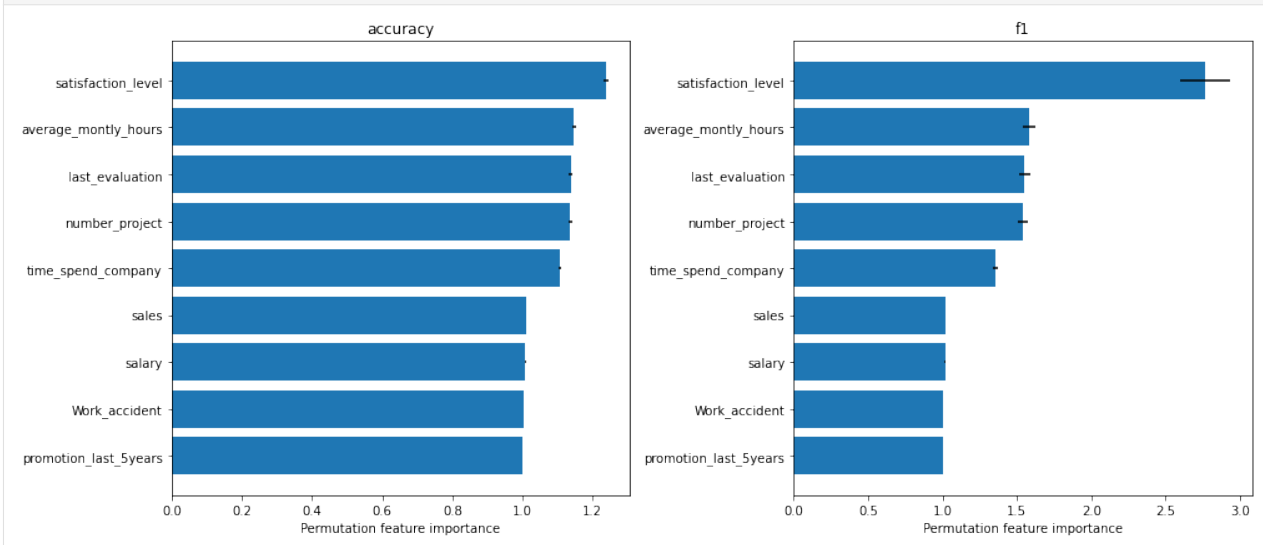
We are now ready to compute the global importance of the features. If the list of features is not provided, by default the explainer will compute the importance of all the features in the dataset. Also, by default the explainer uses the

estimation method to compute the feature importance (i.e., it is less computationally intensive), and the importance returned is the ratio between the original score and the permuted score.

```
[17]: exp = explainer.explain(X=X_test, y=y_test)
100%|=====| 9/9 [00:14<00:00, 1.65s/it]
```

To inspect the results, we can use the built-in `plot_permutation_importance` function.

```
[18]: plot_permutation_importance(exp,
                                n_cols=2,
                                fig_kw={'figwidth': 14, 'figheight': 6});
```



From the F_1 score plot, we can see that the most important feature that the model relies on is the `satisfaction_level`. Following that, we have three features that have approximately the same importance, namely the `average_monthly_hours`, `last_evaluation` and `number_project`. Finally, in our top 5 hierarchy we have `time_spend_company`.

We can observe that features like `sales`, `salary`, `Work_accident` and `promotion_last_5years` receive an importance of 1, which means that they are not relevant for the model (i.e., we are using the ratio between the base and permuted score and a ratio close to 1 means that the original score and the permuted score are approximately the same).

Note that we observe the same ordering in the accuracy plot. It is worth emphasizing that this does not always happen - using a different metric or loss function can give different results.

Custom metrics

We can also use custom score and loss functions apart from the ones already provided. For example, if we would like to use `1 - f1` metric, we need to define the corresponding loss function. To change the metric functions, we need to define a new explainer and ensure that the output of the predictor is compatible with the loss function.

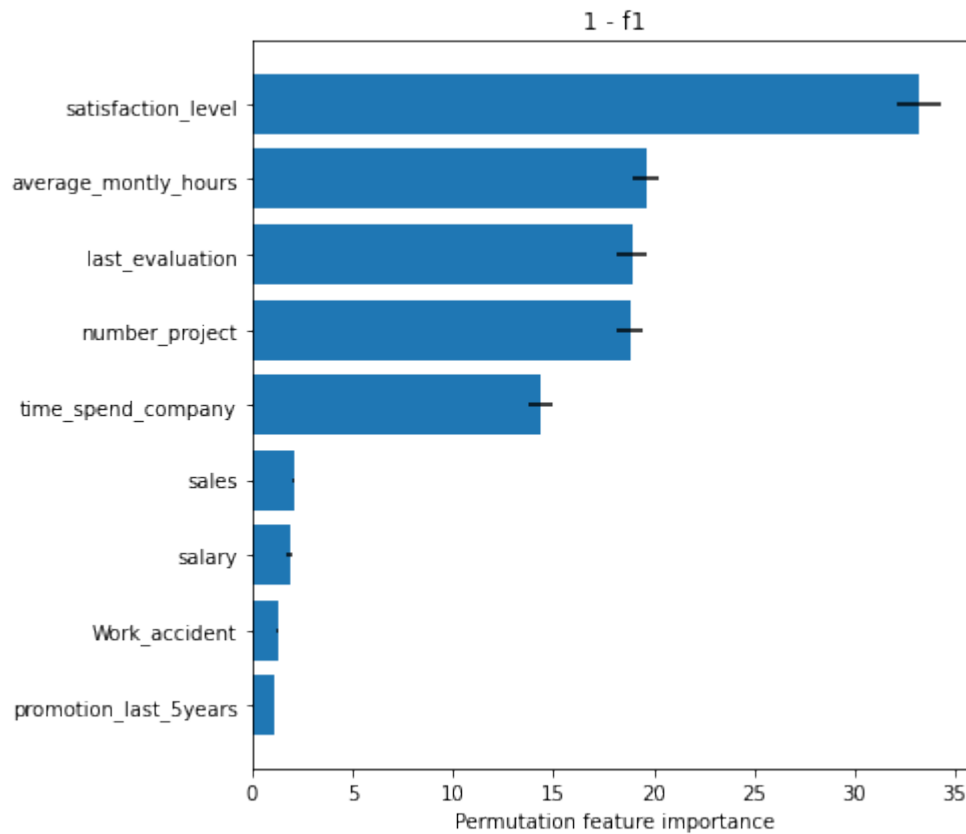
```
[19]: def loss_f1(y_true: np.ndarray, y_pred: np.ndarray) -> float:
      return 1 - f1_score(y_true=y_true, y_pred=y_pred)

[20]: explainer_loss_f1 = PermutationImportance(predictor=predict_fn,
                                              loss_fns={'1 - f1': loss_f1},
                                              feature_names=feature_names,
                                              verbose=True)
```

```
[21]: exp_loss_f1 = explainer_loss_f1.explain(X=X_test, y=y_test)
```

```
100%|=====| 9/9 [00:15<00:00, 1.74s/it]
```

```
[22]: plot_permutation_importance(exp=exp_loss_f1,
                                   fig_kw={'figwidth': 7, 'figheight': 6});
```



As another example, if we want to use the 1 - auc metric we need to define the corresponding loss function, a new predictor which returns the probability of the positive class instead of the label, and implicitly will require to define a new explainer. Note that we need to define a new predictor because the output of the predictor must be compatible with the arguments expected by the loss function.

```
[23]: def loss_auc(y_true: np.ndarray, y_score: np.ndarray) -> float:
        return 1 - roc_auc_score(y_true=y_true, y_score=y_score)
```

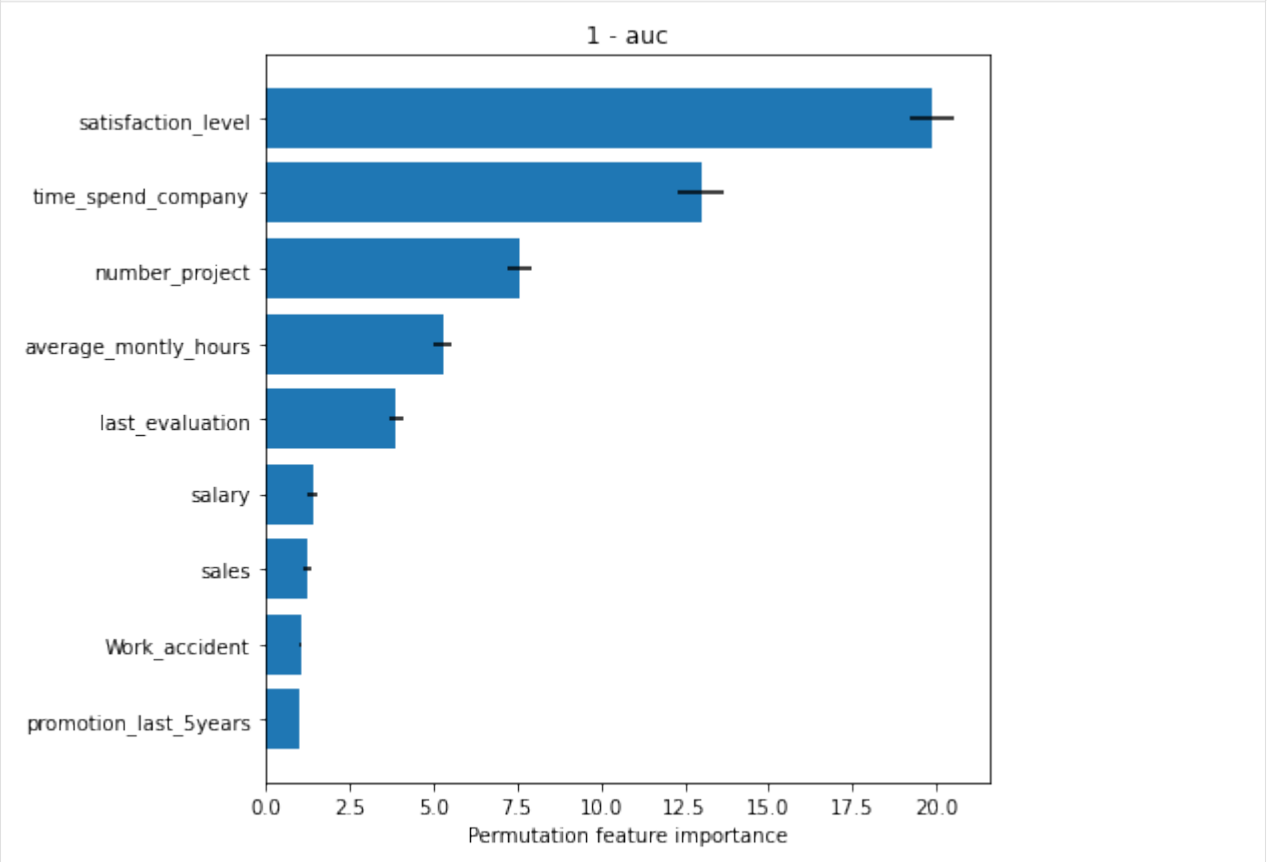
```
def proba_fn(X: np.ndarray) -> np.ndarray:
    return rf.predict_proba(preprocessor.transform(X))[:, 1]
```

```
[24]: explainer_loss_auc = PermutationImportance(predictor=proba_fn,
                                                  loss_fns={'1 - auc': loss_auc},
                                                  feature_names=feature_names,
                                                  verbose=True)
```

```
[25]: exp_loss_auc = explainer_loss_auc.explain(X=X_test, y=y_test)
```

```
100%|=====| 9/9 [00:17<00:00, 1.96s/it]
```

```
[26]: plot_permutation_importance(exp=exp_loss_auc,
                                fig_kw={'figwidth': 7, 'figheight': 6});
```



To conclude, we can observe from the plots above, that the numerical features are quite important for the classification task which agrees with the intuition we developed in our data analysis step. Note that based on the metric used, the importance of the features, and implicitly their ordering can differ. Nevertheless, we observe that the `satisfaction_level` feature is consistently reported as the most important.

References

- [1] Breiman, Leo. “Random forests.” *Machine learning* 45.1 (2001): 5-32.
- [2] Fisher, Aaron, Cynthia Rudin, and Francesca Dominici. “All Models are Wrong, but Many are Useful: Learning a Variable’s Importance by Studying an Entire Class of Prediction Models Simultaneously.” *J. Mach. Learn. Res.* 20.177 (2019): 1-81.

8.13 Similarity explanations

8.13.1 Similarity explanations for 20 newsgroups dataset

In this notebook, we apply the similarity explanation method to a feed forward neural network (FFNN) trained on the 20 newsgroups dataset.

The 20 newsgroups dataset is a corpus of 18846 text documents (emails) divided into 20 sections. The FFNN is trained to classify each document in the correct section. The model uses pre-trained sentence embeddings as input features, which are obtained from raw text using a [pretrained transformer](#).

Given an input document of interest, the similarity explanation method used here aims to find text documents in the training set that are similar to the document of interest according to “how the model sees them”, meaning that the similarity metric makes use of the gradients of the model’s loss function with respect to the model’s parameters.

The similarity explanation tool supports both `pytorch` and `tensorflow` backends. In this example, we will use the `pytorch` backend. Running this notebook on CPU can be very slow, so GPU is recommended.

A more detailed description of the method can be found [here](#). The implementation follows [Charpiat et al., 2019](#) and [Hanawa et al. 2021](#).

```
[ ]: # Installing required sentence transformer
!pip install sentence_transformers
```

```
[1]: import os
import torch
import string
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch.nn as nn
from termcolor import colored
from torch.utils.data import DataLoader
from tqdm import tqdm
from sentence_transformers import SentenceTransformer
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import train_test_split
from alibi.explainers import GradientSimilarity
```

Utils

```
[2]: def to_categorical(y, num_classes):
    """ 1-hot encodes a tensor """
    return np.eye(num_classes, dtype='uint8')[y].astype('float32')

class TorchDataset(torch.utils.data.Dataset):
    """Utility class to create a torch dataloader from numpy arrays.
    """
    def __init__(self, *indexables):
        self.indexables = indexables

    def __getitem__(self, idx):
        output = tuple(indexable[idx] for indexable in self.indexables)
```

(continues on next page)

(continued from previous page)

```

        return output if len(output) > 1 else output[0]

    def __len__(self):
        return len(self.indexables[0])

def append_int(num):
    """Converts an integer into an ordinal (ex. 1 -> 1st, 2 -> 2nd, etc.)

    Parameters
    -----
    num
        Integer number

    Returns
    -----
    Ordinal suffixes
    """
    if num > 9:
        secondToLastDigit = str(num)[-2]
        if secondToLastDigit == '1':
            return 'th'
    lastDigit = num % 10
    if (lastDigit == 1):
        return 'st'
    elif (lastDigit == 2):
        return 'nd'
    elif (lastDigit == 3):
        return 'rd'
    else:
        return 'th'

def break_doc_in_lines(text, nb_words_per_line=18):
    """Breaks document in lines of a fixed number of words for visualization purposes.

    Parameters
    -----
    text
        String to break in line
    nb_words_per_line
        number of words for each line

    Returns
    -----
    String with line breakers
    """
    text_l = text.split(' ')
    text_conc = []
    nb_lines = np.floor(len(text_l) / nb_words_per_line).astype(int) + 1
    for i in range(nb_lines):
        tl = text_l[i * nb_words_per_line: (i + 1) * nb_words_per_line]
        text_conc.append(' '.join(tl))
    text = '\n'.join(text_conc)

```

(continues on next page)

(continued from previous page)

```
return text
```

Load data

Loading and preparing the 20 newsgroups dataset.

Warning:

The cell below will load the 20 news groups dataset. This might require a ↪ considerable amount of memory.

```
[4]: example_idx = 4

print("Loading 20 news groups dataset")
data = fetch_20newsgroups(shuffle=True, random_state=1, subset='train',
                          remove=('headers', 'footers', 'quotes'),
                          return_X_y=False)
X, y = np.asarray(data.data), data.target
target_names = data.target_names
df = pd.DataFrame({'text': X, 'labels': y})

print("Cleaning text")
df['text_cleaned'] = df['text'].str.replace('\s+', ' ')
df['text_cleaned'] = df['text_cleaned'].str.strip()
df['text_cleaned'] = df['text_cleaned'].str.slice(0,131072)
df = df.replace('', np.NaN).dropna()
df = df.drop_duplicates(subset='text_cleaned')
print('')

print(colored("Sample document before cleaning", 'red'))
print(f"{df['text'][example_idx]}")
print('')
print(colored("Sample document after cleaning", 'red'))
print(break_doc_in_lines(f"{df['text_cleaned'][example_idx]}"))
print('')

print("Splitting train - test")
df_train, df_test = train_test_split(df, test_size=0.2)
X_train, y_train = df_train['text_cleaned'].values, df_train['labels'].values
X_test, y_test = df_test['text_cleaned'].values, df_test['labels'].values
y_train, y_test = to_categorical(y_train, num_classes=20), to_categorical(y_test, num_
↪ classes=20)
print(f"X_train shape: {X_train.shape} - y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape} - y_test shape: {y_test.shape}")
```

Loading 20 news groups dataset

Cleaning text

The default value of regex will change from True to False in a future version.

Sample document before cleaning

So its an automatic? Don't know if US spec=CDN spec. for Maximas.

If it is the first set of brake pads on front, then this is fine. My car eats a set every 15k miles or so. The fact that he is replacing the muffler too is also ok.

The mileage is fairly low - but typical fwd stuff is CV joints. Check the maintenance records with the manufacturers requirements for valve adjustments, timing belt changes and so on.

The 60k mile service is often expensive, so make sure he has done everything.

Well, this is one of the commonly cited methods for identifying a car with highway miles.

Might check the gas pedal wear too. Ask him how many sets of tires he has been through. A highway car might have squeezed by on 2 sets, a hard driven car 6-10 sets.

Well, the Maxima should be pretty reliable - but if its out of warranty you should get it checked out by someone knowledgeable first. Stuff for Japanese cars can be expensive.

1995 model year, I believe.

Sample document after cleaning

So its an automatic? Don't know if US spec=CDN spec. for Maximas. If it is the first set of brake pads on front, then this is fine. My car eats a set every 15k miles or so. The fact that he is replacing the muffler too is also ok. The mileage is fairly low - but typical fwd stuff is CV joints. Check the maintenance records with the

↳manufacturers requirements for valve adjustments, timing belt changes and so on. The 60k mile service is often expensive, so, ↳make sure he

has done everything. Well, this is one of the commonly cited methods for identifying a, ↳car with highway

miles. Might check the gas pedal wear too. Ask him how many sets of tires he has been through. A highway car might have squeezed by on 2 sets, a hard driven car 6-10 sets. ↳

↳Well, the Maxima should be pretty reliable - but if its out of warranty you should get it, ↳checked

out by someone knowledgeable first. Stuff for Japanese cars can be expensive. 1995 model, ↳year, I believe.

Splitting train - test

X_train shape: (14604,) - y_train shape: (14604, 20)

(continues on next page)

(continued from previous page)

```
X_test shape: (3652,) - y_test shape: (3652, 20)
```

Define and train model

We define and train a pytorch classifier using sentence embeddings as inputs.

Define model

```
[5]: class EmbeddingModel:
    """Pre-trained sentence transformer wrapper.
    """
    def __init__(
        self,
        model_name: str = 'paraphrase-MiniLM-L6-v2', # https://www.sbert.net/docs/
        ↪ pretrained_models.html
        max_seq_length: int = 200,
        batch_size: int = 32,
        device: torch.device = None
    ) -> None:
        if not isinstance(device, torch.device):
            device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.encode_text = SentenceTransformer(model_name).to(device)
        self.encode_text.max_seq_length = max_seq_length
        self.batch_size = batch_size

    def __call__(self, x: np.ndarray) -> np.ndarray:
        return self.encode_text.encode(x,
                                       convert_to_numpy=True,
                                       batch_size=self.batch_size,
                                       show_progress_bar=False)

class Classifier(nn.Module):
    """FFNN classifier with pretrained sentence embeddings inputs.
    """
    def __init__(
        self,
        n_classes= 20
    ) -> None:
        """ Text classification model from sentence embeddings. """
        super().__init__()
        self.head = nn.Sequential(nn.Linear(384, 256),
                                   nn.LeakyReLU(.1),
                                   nn.Dropout(.5),
                                   nn.Linear(256, n_classes))

    def forward(self, sentence_embeddings) -> torch.Tensor:
        return self.head(sentence_embeddings)
```

```
[6]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Get sentence embeddings and define dataloaders

```
[7]: embedding_model = EmbeddingModel(device=device)

print('Getting train embeddings')
embeddings_train = embedding_model(X_train)
train_loader = DataLoader(TorchDataset(torch.Tensor(embeddings_train).to(device),
                                             torch.Tensor(y_train).to(device)),
                          batch_size=32,
                          shuffle=True)

print('Getting test embeddings')
embeddings_test = embedding_model(X_test)
test_loader = DataLoader(TorchDataset(torch.Tensor(embeddings_test).to(device),
                                             torch.Tensor(y_test).to(device)),
                        batch_size=32,
                        shuffle=False)

Getting train embeddings
Getting test embeddings
```

Train model

```
[8]: epochs = 3

# initialize classifier
model = Classifier().to(device)
print('Training classifier')
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
for epoch in range(epochs):
    for x, y in tqdm(train_loader):
        y_hat = model(x)
        optimizer.zero_grad()
        loss = loss_fn(y_hat, y)
        loss.backward()
        optimizer.step()

Training classifier
100%|=====| 457/457 [00:00<00:00, 972.33it/s]
100%|=====| 457/457 [00:00<00:00, 1189.45it/s]
100%|=====| 457/457 [00:00<00:00, 1195.13it/s]
```

Evaluate model

Evaluating the model on train and test set. Since the dataset is well balanced, we only consider accuracy as evaluation metric.

```
[9]: def eval_model(model, loader):
    model.eval()
    logits, labels = [], []
    with torch.no_grad():
        for x, y in loader:
            y_hat = model(x)
            logits += [y_hat.cpu().numpy()]
            labels += [y.cpu().numpy()]
    logits = np.concatenate(logits, 0)
    preds = np.argmax(logits, 1)
    labels = np.concatenate(labels, 0)
    accuracy = (preds == labels.argmax(axis=1)).mean()
    print(f'Accuracy: {accuracy:.3f}')

print('Train set evaluation')
eval_model(model, train_loader)
print('Test set evaluation')
eval_model(model, test_loader)
```

```
Train set evaluation
Accuracy: 0.720
Test set evaluation
Accuracy: 0.664
```

Find similar instances

Selecting a reference set of 1000 random samples from the training set. The GradientSimilarity explainer will find the most similar instances among those. This downsampling step is performed in order to speed up the fit step.

```
[10]: idxs_ref = np.random.choice(len(X_train), 1000, replace=False)
X_ref = X_train[idxs_ref]
embeddings_ref = embeddings_train[idxs_ref]
y_ref = y_train[idxs_ref]
```

Initializing a GradientSimilarity explainer instance.

```
[11]: gsm = GradientSimilarity(model,
                               loss_fn,
                               precompute_grads=True,
                               sim_fn='grad_cos',
                               backend='pytorch',
                               device=device)
```

Fitting the explainer on the reference data.

```
[12]: gsm.fit(embeddings_ref, y_ref)
```

```
[12]: GradientSimilarity(meta={
    'name': 'GradientSimilarity',
    'type': ['whitebox'],
    'explanations': ['local'],
    'params': {
        'sim_fn_name': 'grad_cos',
        'store_grads': True,
        'backend_name': 'pytorch',
        'task_name': 'classification'
    },
    'version': '0.6.6dev'
})
```

Selecting 3 random instances from the test set. We only select documents with less than 1000 characters for visualization purposes.

```
[13]: idxs_samples = np.where(np.array([len(x) for x in X_test]) <= 1000)[0]
    idxs_samples = np.random.choice(idxs_samples, 3, replace=False)

    X_sample, embeddings_sample, y_sample = X_test[idxs_samples], embeddings_test[idxs_
    ↪samples], y_test[idxs_samples]
```

Getting predictions and explanations for each of the 5 test samples.

```
[14]: preds = model(torch.Tensor(embeddings_sample).to(device)).detach().cpu().numpy().
    ↪argmax(axis=1)
    expls = gsm.explain(embeddings_sample, y_sample)
```

Visualizations

Building a dictionary for each sample for visualization purposes. Each dictionary contains:

- The original text document `x` (not the embedding representation).
- The corresponding label `y`.
- The corresponding model's prediction `pred`.
- The reference instances ordered by similarity `X_sim`.
- The corresponding reference labels ordered by similarity `y_sim`.
- The corresponding model's predictions for the reference set `preds_sim`.

```
[15]: ds = []
    for j in range(len(embeddings_sample)):
        y_sim = y_ref[expls.data['ordered_indices'][j]].argmax(axis=1)
        X_sim = X_ref[expls.data['ordered_indices'][j]]
        sim_embedding = embeddings_ref[expls.data['ordered_indices'][j]]
        preds_sim = model(torch.Tensor(sim_embedding).to(device)).detach().cpu().numpy().
        ↪argmax(axis=1)

        d = {'x': X_sample[j],
            'y': y_sample[j].argmax(),
            'pred': preds[j],
```

(continues on next page)

(continued from previous page)

```

        'X_sim': X_sim,
        'y_sim': y_sim,
        'preds_sim': preds_sim}
ds.append(d)

```

Most similar instances

Showing the 3 most similar instances for each of the test instances.

```

[16]: for sample_nb in range(3):
        title = f"Sample nb {sample_nb}"
        print(colored(title, 'blue'))
        print(colored(f"{len(title) * '='}", 'blue'))
        print('')

        print(colored("Original instance - ", 'red'),
              colored(f"Label: {target_names[ds[sample_nb]['y']]}", 'red'),
              colored(f"Prediction: {target_names[ds[sample_nb]['pred']]}", 'red'))
        print(break_doc_in_lines(f"{ds[sample_nb]['x']}"))
        print('')

        for i in range(3):
            print(colored(f"{i+1}{append_int(i+1)} most similar instance - ", 'red'),
                  colored(f"Label: {target_names[ds[sample_nb]['y_sim'][i]]}", 'red'),
                  colored(f"Prediction: {target_names[ds[sample_nb]['preds_sim'][i]]}", 'red')
            →))
            print(break_doc_in_lines(f"{ds[sample_nb]['X_sim'][i]}"))
            print('')

```

Sample nb 0

=====

Original instance - Label: misc.forsale - Prediction: misc.forsale

One pair of kg1's in Oak finish with black grilles. Includes original packaging. \$200 +
 →shipping Firm.

1st most similar instance - Label: misc.forsale - Prediction: misc.forsale

Kirsch Pull down Window Shades - White, Light Filtering - 73.25" Wide, 72" High, can be
 →cut to
 width - Brand new, unopened - "Best Quality", Vinyl Coated Cotton - Mounting Brackets.
 →included - \$35 (Bought
 at \$60 at J.C.Penney)

2nd most similar instance - Label: misc.forsale - Prediction: misc.forsale

Hey, collection. I am interested in buying any in good condition. I am particularly
 →interested in any of
 the older, exotic models (eg five] transformers into one etc... I am looking at paying
 →around \$20-\$40
 depending upon the model, size and original cost etc. I will also pay airmail postage.
 →and packing. I
 am also happy to buy any old sci-fi related toys eg robots, rocketships, micronauts.

(continues on next page)

(continued from previous page)

→etc... There is only
 one catch. I live in New Zealand so you have to be willing to post the items there.
 I hop that someone can help me out. Cheers

3rd most similar instance - Label: misc.forsale - Prediction: misc.forsale
 HP 9872B 4 pen plotter. \$150 Fujistu M2451E 130 meg SCSI tape drive \$150 Sony 40 meg SCSI
 disk drive (sticks once in a while) \$50 Dead Maxtor XT4380E 338 meg ESDI drive \$100 Dead
 →Miniscribe
 20 meg SCSI drive \$10 Adaptac SCSI to ST-412 interface board \$20 Daughter boards from
 →tape drives ?QIC-02
 - QIC-36? \$20 Twist Terms (VT100 terms that the head twists on for 80x25 or 80x72) \$150
 →14"
 Analog RGB color monitor (15.7 Khz works nice with amiga's) \$100 Spool with 90+ feet of
 →50 conductor
 ribbon cable \$75 All prices are or best offer. Prices do not include UPS shipping. All
 →items working
 except those stated as Dead.

Sample nb 1

=====

Original instance - Label: sci.crypt - Prediction: sci.crypt
 I am looking for some Public Domain (and exportable) code for encryption. Nothing
 →elaborate, just something that will
 satisfy a marketing need :-) Oh yes, UNIX platform.

1st most similar instance - Label: sci.crypt - Prediction: sci.crypt
 Hmmm. I think, with really large keyspaces like this, you need to alter the strategy
 →discussed for DES.
 Attempt decryption of several blocks, and check the disctribution of the contents. I don
 →'t think it's at all
 feasible to keep 2**80 encryptions of a known plaintext block on *any* amount of tape or
 →CD-ROM. And
 certainly not 2**128 such encrypted blocks. (Anyone know a cheap way of converting every
 →atom in the solar
 system into a one bit storage device?) Actually, a keysearch of this kind shouldn't be
 →much worse than
 the simpler kind in terms of speed. It's just that you have to do it over for *every*
 encrypted message. Dumb question: Has anyone ever done any serious research on how many
 →legitimate ASCII-encoded 8-byte blocks
 there are that could be part of an english sentence? For attacking DES in ECB mode, it
 →seems
 like a dictionary of this kind might be pretty valuable...

2nd most similar instance - Label: sci.crypt - Prediction: sci.crypt
 As am I If "high quality secure NSA classified technology" means handing my key over to
 →whomever, I'll
 take PGP any day. Right now they are billing it as voluntary, i.e. bend over, here it
 →comes.
 As soon as enough Wiretap chip based units are out there, how much easier do you think it
 will be to redefine "on your own" to mean write it yourself and don't even THINK about
 →distributing

(continues on next page)

(continued from previous page)

it...? Get honest, no one is going to buy this trash if they KNOW it's compromised.
 ↪ already, and
 less will buy it if the algorithm is not disclosed. The NSA knows that making this stuff
 ↪ available
 to the public means handing it to whatever foreign powers are interested in the process.
 ↪ Since when has
 export control stopped anyone (especially software wise) Ask yourself carefully if "
 ↪ high quality secure NSA classified technology
 " is something they are going to hand out. Not unless you can drive a NSA van through
 the holes. uni (Dark)

3rd most similar instance - Label: sci.crypt - Prediction: sci.crypt

You're reading far too much into this (aside from the obvious fact that you shouldn't
 ↪ hold anybody to
 what they wrote in a 10 year old book in a rapidly changing field like this.) Quite
 ↪ simply
 she says that the security should not DEPEND on the secrecy of the algorithm. A secret
 ↪ algorithm can
 still be secure, after all, we just don't know it. Only our level of trust is affected,
 ↪ not
 the security of the system. The algorithm *could* be RSA for all we know, which we
 ↪ believe to
 be secure. They have a much better reason to classify the algorithm than to protect its
 ↪ security. They
 want to protect its market share. If they publish the algorithm, then shortly
 ↪ manufacturers would make chips that
 implement the algorithm and standard but do not use a key stored in escrow. And of
 ↪ course, everybody
 would buy them. The whole push of this chip is that by establishing a standard that you
 ↪ can
 only use if you follow their rules, they get us to follow their rules without enacting
 ↪ new laws
 that we would fight tooth and nail. Quite simply, with Clipper established, it would be
 ↪ much harder for
 another encryption maker to define a new standard, to make phones that can't talk to the
 ↪ leading phone
 companies. The result is tappable cryptography without laws forbidding other kinds, for
 ↪ 99% of the populace. To get
 untappable crypto, you would have to build a special phone that runs on top of this
 ↪ system, and
 everybody you talk to would have to have an indentical one. That's the chicken and egg
 ↪ of crypto.
 The government is using its very special ability to solve chicken and egg problems of
 ↪ new technologies to
 control this one in a way they like. It's almost admirably clever. When the EFF started,
 ↪ I posed
 the question here "What are the police going to do when they wake up and discover they
 ↪ can't
 wiretap?" and nobody here had an answer (or even thought it was much of a question) Then
 ↪ came
 the backdoor and Digital Telephony bills, which we fought. Now we have their real answer,
 ↪ the cleverest of

(continues on next page)

(continued from previous page)

all.

Sample nb 2

=====

Original instance - Label: comp.graphics - Prediction: comp.graphics

Sorry if this is a FAQ but : "Where can I get a 286 (16 bit) version of
POV-Ray ? " Any help would be greatly appreciated. I need the 286 version since Turbo_

↳Pascal won't

let me run a 32 bit program from within my program. Any info on this would also be
a great help. Thanks, Byron. bkidd@esk.compserv.utas.edu.au B.Kidd@cam.compserv.utas.edu.

↳au --

1st most similar instance - Label: comp.graphics - Prediction: comp.graphics

FOR IMMEDIATE RELEASE Editorial Contact: Single Source Marketing: Myra Manahan (714) 545-

↳1338 Genoa Systems: Joseph Brunoli (408) 432-9090

Neil Roehm (408) 432-9090/Technical Genoa Presents High Performance Video Graphics_

↳Accelerator SAN JOSE, Calif USA -- Genoa Systems

Corporation announces WINDOWSVGA 24, a True Color 24-bit graphics accelerator card that_

↳delivers up to 16.8 million colors

at speeds faster than the competition. Plus it offers a full range of resolutions, high_

↳refresh rates as

well as unique proprietary performance features. The card is available in both 16-bit_

↳ISA bus and 32-bit VESA

Local bus versions (models 8500 AND 8500VL). With 1MB DRAM on board, the WINDOWSVGA 24_

↳card offers maximum

resolution up to 1,280 x 1,024 and supports a refresh rate of 72Hz at 800 x 600 and

resolution up to 1,024 x 768 non-interlaced. Both models provide performance many times_

↳greater than standard SVGA boards,

yet conform to all current video standards. WINDOWSVGA 24 features Genoa's_

↳FlickerFree(tm) technology, which eliminates screen flash and

flicker to make viewing much more comfortable. the cards also come with Safescan(tm), a_

↳utility developed by Genoa

to eliminate the black border around the screen and thereby provide 100-percent screen_

↳use for overscanning monitors. WINDOWSVGA

model 8500VL takes full advantage of the speed offered by the new VESA Local bus_

↳technology. Most VL

bus cards will only handle data transfers up to 33MHz, but the 8500VL will transfer data_

↳at the

full speed of the CPU, up to 50MHz. Genoa is also offering this card in the "TurboBahn"_

↳combination

packaged with their TURBOEXPRESS 486VL motherboard. Built around the Cirrus Logic GD-

↳5426 GUI accelerator, WINDOWSVGA 24 offers the

user an exceptional price/performance value. Genoa's advanced proprietary drivers act to

↳"turbocharge" the chip, thereby providing an affordable

accelerator card with power and performance that surpass many of the more highly priced_

↳chip cards. The Genoa

user will enjoy optimal speed and reliability for such programs as Windows, AutoCAD,_

↳AutoShade, 3D Studio, OS/2, OrCAD

and more. Driver updates and product bulletins are available on Genoa's BBS at (408) 943-

↳1231. Genoa Systems manufactures

and markets an extensive line of graphics adapters, motherboards, audio and multimedia_

(continues on next page)

(continued from previous page)

→ cards for IBM-compatible personal computers. All products come with a two year limited warranty on parts and labor. Genoa products are currently distributed worldwide through authorized distributors, resellers, VARs and systems integrators. For more information contact Joe Brunoli, Marketing Manager, Genoa Systems at 75 E. Trimble Road, San Jose, Calif. 95131; Tel: (408) 432-9090 or (800) 934-3662; → Fax: (408) 434-0997.

2nd most similar instance - Label: comp.graphics - Prediction: comp.graphics

Well, the temp file thing creates an obvious problem: it is impossible to use cvview for viewing CD-ROM based picture collections. And it is the ONLY non- windows viewer that works properly with my Cirrus-based 24 bit VGA.

3rd most similar instance - Label: comp.graphics - Prediction: comp.graphics

If you are looking for viewer try VPIC60

Most similar labels distributions

Showing the average similarity scores for each group of instances in the reference set belonging to the same true class and to the same predicted class.

```
[17]: def plot_distributions(ds, expls, target_names, figsize=(20, 5)):

    for i in range(len(ds)):
        fig, axes = plt.subplots(1, 2, figsize=figsize, sharex=False)
        d = ds[i]

        y_sim = d['y_sim']
        preds_sim = d['preds_sim']
        y = d['y']
        pred = d['pred']
        df_distribution = pd.DataFrame({'y_sim': y_sim,
                                      'preds_sim': preds_sim,
                                      'scores': expls.data['scores'][i]})

        title = f"Sample nb {i}"
        print(colored(title, 'blue'))
        print(colored(f"{len(title) * '='}", 'blue'))
        print('')

        print(colored("Original instance", 'red'))
        print(colored(f"Label: section {d['y']}", {target_names[d['y']]}, 'red'))
        print(colored(f"Prediction: section {d['pred']}", {target_names[d['pred']]}, 'red')
        → ))
        print(break_doc_in_lines(f"{d['x']}"))

        df_y = df_distribution.groupby('y_sim')['scores'].mean()
        df_y.index = target_names
```

(continues on next page)

(continued from previous page)

```

df_y.sort_values(ascending=True).plot(kind='barh', ax=axes[0])
axes[0].set_title("Averaged scores for each true class in reference set \n")

df_preds = df_distribution.groupby('preds_sim')['scores'].mean()
df_preds.index = target_names
df_preds.sort_values(ascending=True).plot(kind='barh', ax=axes[1])
axes[1].set_title("Averaged scores for each predicted class in reference set \n")
fig.tight_layout()
plt.show()

```

[19]: plot_distributions(ds, expls, target_names)

Sample nb 0

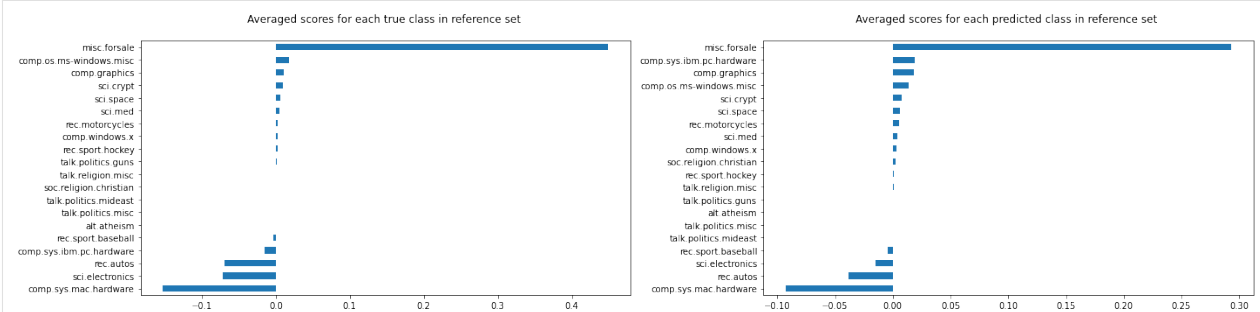
=====

Original instance

Label: section 6, misc.forsale

Prediction: section 6, misc.forsale

One pair of kg1's in Oak finish with black grilles. Includes original packaging. \$200 +
 ↳shipping Firm.



Sample nb 1

=====

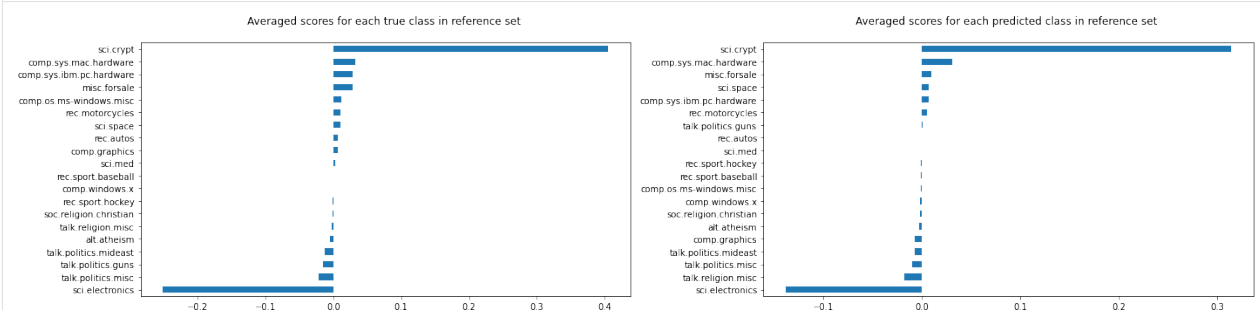
Original instance

Label: section 11, sci.crypt

Prediction: section 11, sci.crypt

I am looking for some Public Domain (and exportable) code for encryption. Nothing.

↳elaborate, just something that will satisfy a marketing need :-). Oh yes, UNIX platform.



Sample nb 2

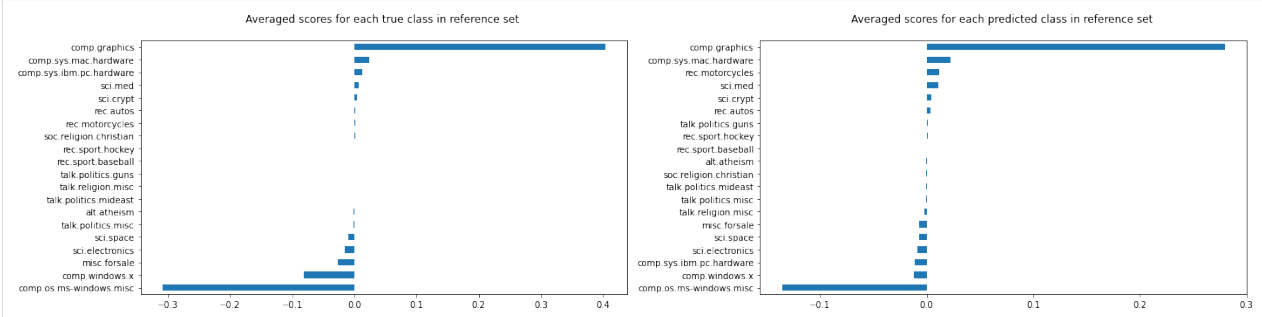
=====

(continues on next page)

(continued from previous page)

Original instance**Label:** section 1, comp.graphics**Prediction:** section 1, comp.graphics

Sorry if this is a FAQ but : "Where can I get a 286 (16 bit) version of
POV-Ray ? " Any help would be greatly appreciated. I need the 286 version since Turbo_↵
↵Pascal won't
let me run a 32 bit program from within my program. Any info on this would also be
a great help. Thanks, Byron. bkidd@esk.compserv.utas.edu.au B.Kidd@cam.compserv.utas.edu.
↵au --



The plots show how the instances belonging to the same class (and the instances classified by the model as belonging to the same class) of the instance of interest have on average higher similarity scores, as expected.

8.13.2 Similarity explanations for ImageNet

In this notebook, we apply the similarity explanation method to a ResNet model pre-trained on the ImageNet dataset. We use a subset of the ImageNet dataset including 1000 random samples as training set for the explainer. The training set is constructed by picking 100 random images for each of the following classes:

- ‘stingray’
- ‘trilobite’
- ‘centipede’
- ‘slug’
- ‘snail’
- ‘Rhodesian ridgeback’
- ‘beagle’
- ‘golden retriever’
- ‘sea lion’
- ‘espresso’

The test set contains 50 random samples, 5 for each of the classes above. The data set is stored in a public google storage bucket and can be fetched using the utility function `fetch_imagenet_10`.

Given an input image of interest picked from the test set, the similarity explanation method used here aims to find images in the training set that are similar to the image of interest according to “how the model sees them”, meaning that the similarity metric makes use of the gradients of the model’s loss function with respect to the model’s parameters.

The similarity explanation tool supports both `pytorch` and `tensorflow` backends. In this example, we will use the `tensorflow` backend. Running this notebook on CPU can be very slow, so GPU is recommended.

A more detailed description of the method can be found [here](#). The implementation follows Charpiat et al., 2019 and Hanawa et al. 2021.

```
[4]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.metrics import accuracy_score
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.losses import categorical_crossentropy
from alibi.datasets import fetch_imagenet_10
from alibi.explainers import GradientSimilarity
```

Utils

```
[18]: def plot_similar(ds, expls, int_to_str, mean_channels, figsize=(20, 20)):
    """Plots original instances and similar instances.

    Parameters
    -----
    ds
        List of dictionaries containing instances to plot, labels and predictions.
    expls
        Similarity explainer explanation object.
    int_to_str
        Dictionary mapping label's number to label's names.
    mean_channels
        Mean channels to add to the images for visualization.
    figsize
        Figure size.

    Returns
    -----
    None
    """
    fig, axes = plt.subplots(5, 6, figsize=figsize, sharex=False)
    for j in range(len(ds)):
        d = ds[j]
        axes[j, 0].imshow(BGR_to_RGB(d['x']))
        label_orig = int_to_str[d['y']].split(',')[0]
        if len(label_orig) > 16:
            label_orig = label_orig[:13] + '...'
        pred_orig = int_to_str[d['pred']].split(',')[0]
        if len(pred_orig) > 16:
            pred_orig = pred_orig[:13] + '...'
        if j == 0:
            title_orig = "Original instance"
            axes[j, 0].set_title(f"{title_orig} \n" +
```

(continues on next page)

(continued from previous page)

```

        f"{len(title_orig) * '='} \n" +
        f"Label: {label_orig} \n" +
        f"Prediction: {pred_orig} ")

    else:
        axes[j, 0].set_title(f"Label: {label_orig} \n" +
                             f"Prediction: {pred_orig} ")
    axes[j, 0].axis('off')
    for i in range(expls.data['most_similar'].shape[0]):
        label_sim = int_to_str[d['y_sim'][i]].split(',')[0]
        if len(label_sim) > 16:
            label_sim = label_sim[:13] + '...'
        pred_sim = int_to_str[d['preds_sim'][i]].split(',')[0]
        if len(pred_sim) > 16:
            pred_sim = pred_sim[:13] + '...'
        most_similar = BGR_to_RGB((expls.data['most_similar'][j] + mean_channels).
↪ astype(int)[i])

        axes[j, i + 1].imshow(most_similar, cmap='gray')
        if j == 0:
            title_most_sim = f"{i+1}{append_int(i+1)} most similar instance"
            axes[j, i + 1].set_title(f"{title_most_sim} \n" +
                                     f"{len(title_most_sim) * '='} \n"+
                                     f"Label: {label_sim} \n" +
                                     f"Prediction: {pred_sim}")
        else:
            axes[j, i + 1].set_title(f"Label: {label_sim} \n" +
                                     f"Prediction: {pred_sim}")
        axes[j, i + 1].axis('off')

plt.show()

def plot_distributions(ds, expls, int_to_str, figsize=(20, 20)):
    """Plots original instances and scores distributions per class.

    Parameters
    -----
    ds
        List of dictionaries containing instances to plot, labels and predictions.
    expls
        Similarity explainer explanation object.
    int_to_str
        Dictionary mapping label's number to label's names.
    figsize
        Figure size.

    Returns
    -----
    None
    """

```

(continues on next page)

(continued from previous page)

```

fig, axes = plt.subplots(5, 2, figsize=figsize, sharex=False)

for i in range(len(ds)):
    d = ds[i]

    y_sim = d['y_sim']
    preds_sim = d['preds_sim']
    y = d['y']
    pred = d['pred']
    df_ditribution = pd.DataFrame({'y_sim': y_sim,
                                   'scores': expls.data['scores'][i]})

    axes[i, 0].imshow(BGR_to_RGB(d['x']))
    if i == 0:
        title_orig = "Original instance"
        axes[i, 0].set_title(f"{title_orig} \n " +
                             f"{len(title_orig) * '='} \n" +
                             f"Label: {d['y']} - {int_to_str[d['y']]} \n" +
                             f"Prediction: {d['pred']} - {int_to_str[d['pred']]}.
↪split(', ')[0]} ")
    else:
        axes[i, 0].set_title(f"Label: {d['y']} - {int_to_str[d['y']]}
↪split(', ')[0]} "
        axes[i, 0].axis('off')
    df_y = df_ditribution.groupby('y_sim')['scores'].mean()
    df_y.index = [int_to_str[i] for i in df_y.index]
    df_y.sort_values(ascending=True).plot(kind='barh', ax=axes[i, 1])
    if i == 0:
        title_true_class = "Averaged scores for each true class in reference set"
        axes[i, 1].set_title(f"{title_true_class} \n " +
                             f"{len(title_true_class) * '='} \n ")

fig.tight_layout()
plt.show()

def append_int(num):
    """Converts an integer into an ordinal (ex. 1 -> 1st, 2 -> 2nd, etc.).

    Parameters
    -----
    num
        Integer number.

    Returns
    -----
    Ordinal suffixes.
    """
    if num > 9:
        secondToLastDigit = str(num)[-2]
        if secondToLastDigit == '1':

```

(continues on next page)

(continued from previous page)

```

        return 'th'
    lastDigit = num % 10
    if (lastDigit == 1):
        return 'st'
    elif (lastDigit == 2):
        return 'nd'
    elif (lastDigit == 3):
        return 'rd'
    else:
        return 'th'

def subtract_mean_channel(X):
    """Subtracts the mean channels from a batch of images.

    Parameters
    -----
    X
        Batches of images to subtract the mean channel from.
    Returns
    -----
    Batch of images.
    """
    assert len(X.shape) == 4
    mean_channels = np.array([103.939, 116.779, 123.68]).reshape(1, 1, 1, -1)
    X_mean = X - mean_channels
    return X_mean, mean_channels

def BGR_to_RGB(X):
    if len(X.shape) == 4:
        return X[:, :, :, ::-1]
    elif len(X.shape) == 3:
        return X[:, :, ::-1]
    else:
        raise ValueError('Incorrect shape')

```

Load data

Fetching and preparing the reduced ImageNet dataset.

```
[6]: imagenet10 = fetch_imagenet_10()
```

```
[7]: X_train, y_train = imagenet10['trainset']
    X_train, mean_channels = subtract_mean_channel(X_train)
    X_test, y_test = imagenet10['testset']
    X_test, _ = subtract_mean_channel(X_test)
    int_to_str = imagenet10['int_to_str_labels']
    y_train = to_categorical(y_train, num_classes=1000)
    y_test = to_categorical(y_test, num_classes=1000)

```



```
[8]: i = 0
label = y_train.argmax(axis=1)[i]
print(f"Label: {label} - {int_to_str[label]}")
x = BGR_to_RGB((X_train + mean_channels).astype(int)[i])

plt.imshow(x);
plt.axis('off');

Label: 6 - stingray
```



Load model

Load a pretrained tensorflow model with a ResNet architecture trained on the ImageNet dataset.

```
[ ]: model = ResNet50(weights='imagenet')
preds = model(X_test).numpy().argmax(axis=1)
acc = accuracy_score(y_test.argmax(axis=1), preds)
```

```
[10]: print('Test accuracy: ', acc)

Test accuracy:  0.86
```

Find similar instances

Initializing a GradientSimilarity explainer instance.

```
[11]: gsm = GradientSimilarity(model, categorical_crossentropy, precompute_grads=False, sim_fn=
    ↪ 'grad_cos')
```

Fitting the explainer on the training data.

```
[12]: gsm.fit(X_train, y_train)

[12]: GradientSimilarity(meta={
    'name': 'GradientSimilarity',
    'type': ['whitebox'],
    'explanations': ['local'],
```

(continues on next page)

(continued from previous page)

```
'params': {
    'sim_fn_name': 'grad_cos',
    'store_grads': False,
    'backend_name': 'tensorflow',
    'task_name': 'classification'}

'version': '0.6.6dev'}
)
```

Selecting 5 random classes out of 10 and 1 random instance per class from the test set (5 test instances in total).

```
[13]: idxs_samples = np.array([np.random.choice(range(5 * i, 5 * i + 5)) for i in range(10)])
idxs_samples = np.random.choice(idxs_samples, 5, replace=False)

X_sample, y_sample = X_test[idxs_samples], y_test[idxs_samples]
preds = model(X_sample).numpy().argmax(axis=1)
```

Getting the most similar instance for the each of the 5 test samples.

```
[14]: expls = gsm.explain(X_sample, y_sample)
```

Visualizations

Building a dictionary for each sample for visualization purposes. Each dictionary contains

- The original image `x` (with mean channels added back for visualization).
- The corresponding label `y`.
- The corresponding model's prediction `pred`.
- The corresponding reference labels ordered by similarity `y_sim`.
- The corresponding model's predictions for the reference set `preds_sim`.

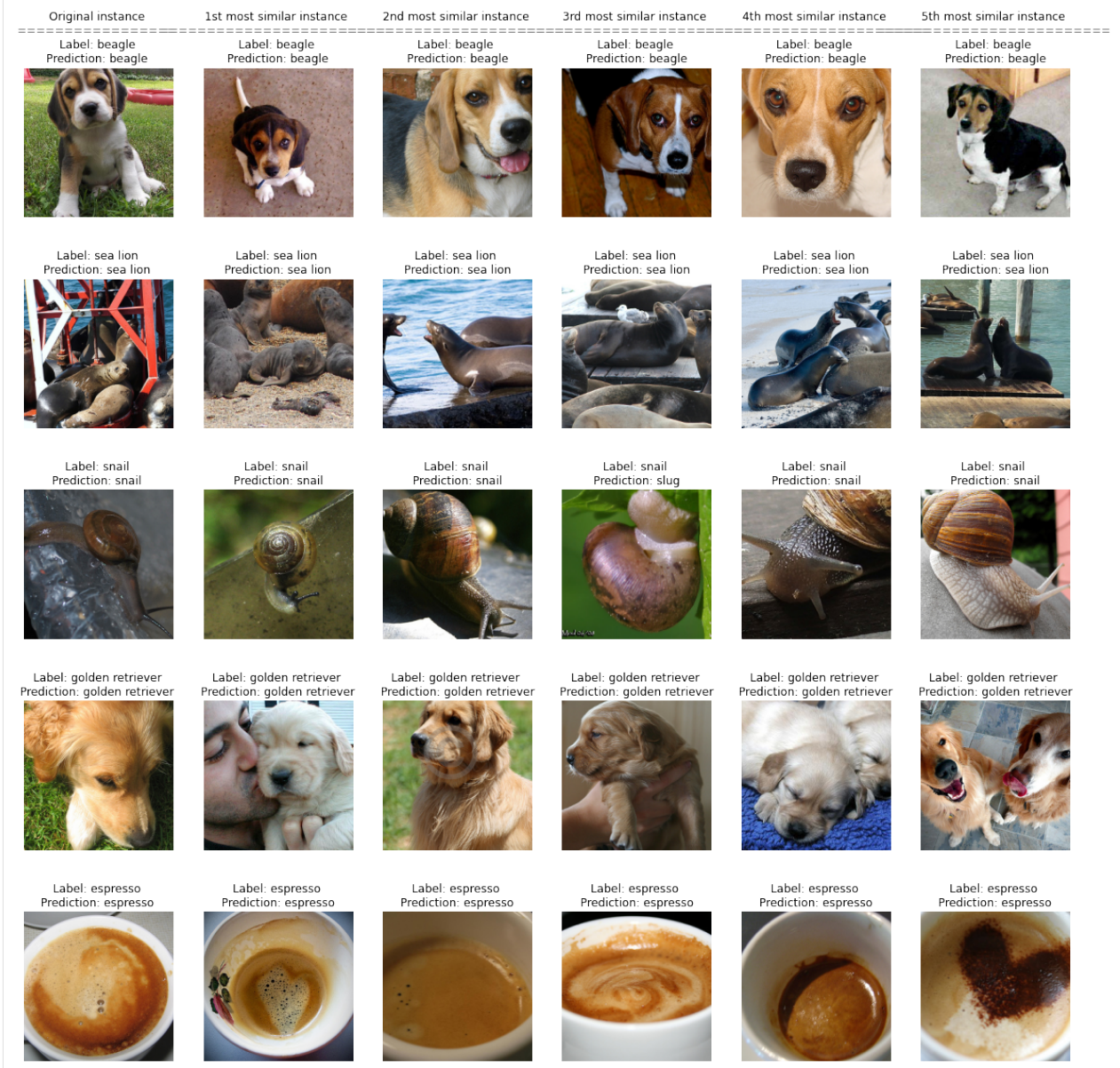
```
[15]: ds = []
for j in range(len(X_sample)):
    y_sim = y_train[expls.data['ordered_indices'][j]].argmax(axis=1)
    X_sim = X_train[expls.data['ordered_indices'][j][:5]]
    preds_sim = model(X_sim).numpy().argmax(axis=1)

    d = {'x': (X_sample + mean_channels).astype(int)[j],
        'y': y_sample[j].argmax(),
        'pred': preds[j],
        'y_sim': y_sim,
        'preds_sim': preds_sim}
    ds.append(d)
```

Most similar instances

Showing the 5 most similar instances for each of the test instances, ordered from the most similar to the least similar.

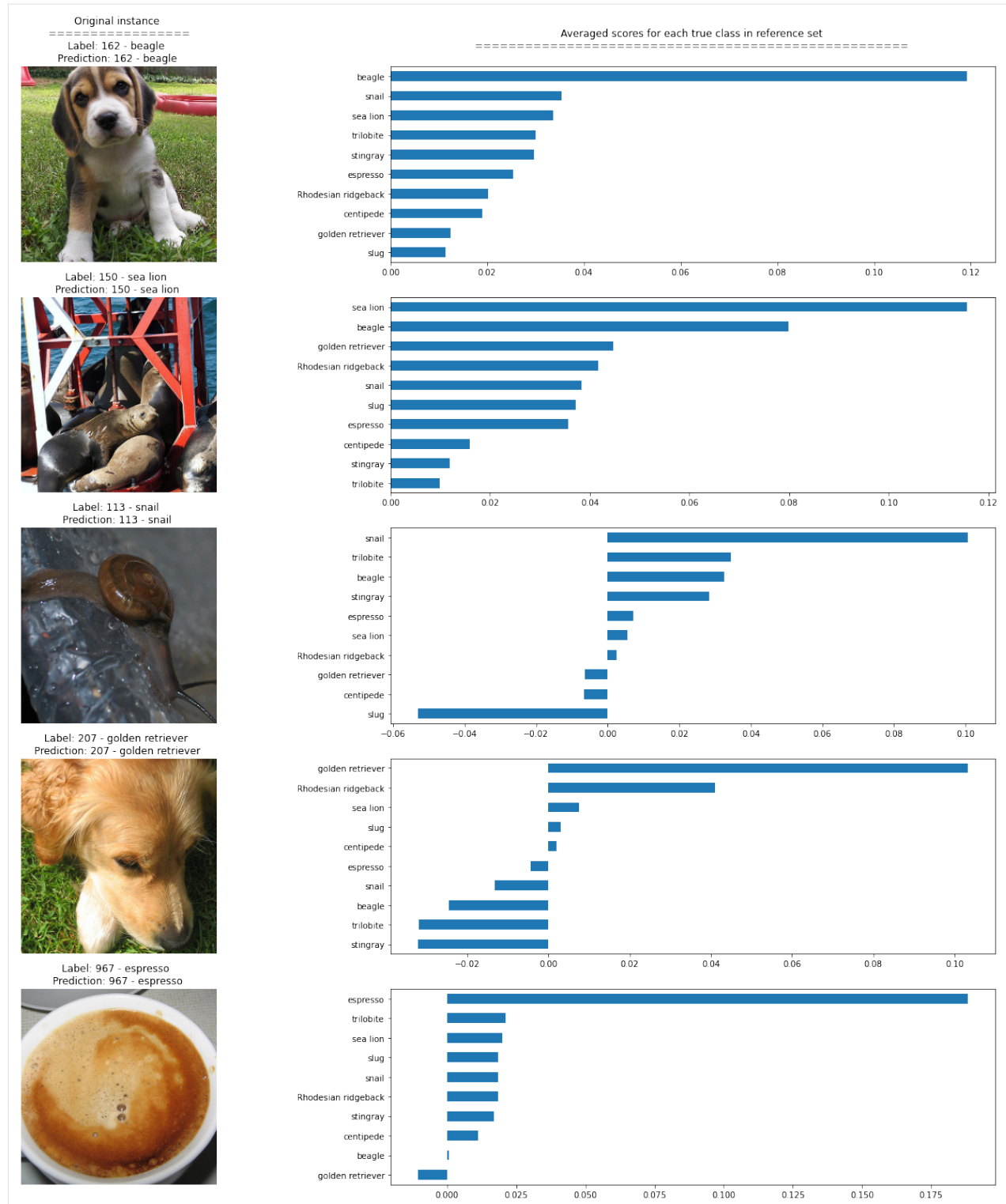
```
[16]: plot_similar(ds, expls, int_to_str, mean_channels)
```



Most similar labels distributions

Showing the average similarity scores for each group of instances in the reference set belonging to the same true class. It can be seen that the higher score corresponds to the class of the original instance, as expected.

```
[19]: plot_distributions(ds, expls, int_to_str)
```



8.13.3 Similarity explanations for MNIST

In this notebook, we apply the similarity explanation method to a convolutional network trained on the MNIST dataset. Given an input image of interest, the similarity explanation method used here aims to find images in the training dataset that are similar to the image of interest according to “how the model sees them”, meaning that the similarity metric makes use of the gradients of the model’s loss function with respect to the model’s parameters. The explanation should be interpreted along the line of “I classify this image as a 4 because I find it similar to another image in the training set that was labeled as a 4.”

The similarity explanation tool supports both `pytorch` and `tensorflow` backends. In this example, we will use the `tensorflow` backend.

A more detailed description of the method can be found [here](#). The implementation follows Charpiat et al., 2019 and Hanawa et al. 2021.

```
[1]: import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.layers import Activation, Conv2D, Dense, Dropout
from tensorflow.keras.layers import Flatten, Input, Reshape, MaxPooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.losses import categorical_crossentropy
from alibi.explainers import GradientSimilarity
```

Utils

```
[2]: def plot_similar(ds, expls, figsize=(20, 20)):
    """Plots original instances and similar instances.

    Parameters
    -----
    ds
        List of dictionaries containing instances to plot, labels and predictions.
    expls
        Similarity explainer explanation object.
    figsize
        Figure size.

    Returns
    -----
    None
    """
    fig, axes = plt.subplots(5, 6, figsize=figsize, sharex=False)
    for j in range(len(ds)):
        d = ds[j]
        axes[j, 0].imshow(np.squeeze(d['x']), cmap='gray')
        axes[j, 0].axis('off')
        if j == 0:
            title_orig = "Original instance"
            axes[j, 0].set_title(f"{title_orig} \n" +
```

(continues on next page)

(continued from previous page)

```

                                f"{len(title_orig) * '='} \n" +
                                f"Label: {d['y']} - Prediction: {d['pred']} ")
    else:
        axes[j, 0].set_title(f"Label: {d['y']} - Prediction: {d['pred']} ")
    for i in range(expls.data['most_similar'].shape[0]):
        most_similar = np.squeeze(expls.data['most_similar'][j][i])
        axes[j, i + 1].imshow(most_similar, cmap='gray')
        axes[i, i + 1].axis('off')
        if j == 0:
            title_most_sim = f"{i+1}{append_int(i+1)} most similar instance"
            axes[j, i + 1].set_title(f"{title_most_sim} \n" +
                                    f"{len(title_most_sim) * '='} \n" +
                                    f"Label: {d['y_sim'][i]} - Prediction: {d[
↪ 'preds_sim'][i]}")
        else:
            axes[j, i + 1].set_title(f"Label: {d['y_sim'][i]} - Prediction: {d[
↪ 'preds_sim'][i]}")

    plt.show()

def plot_distributions(ds, expls, figsize=(20, 20)):
    """Plots original instances and scores distributions per class.

    Parameters
    -----
    ds
        List of dictionaries containing instances to plot, labels and predictions.
    expls
        Similarity explainer explanation object.
    figsize
        Figure size.

    Returns
    -----
    None
    """

    fig, axes = plt.subplots(5, 3, figsize=figsize, sharex=False)

    for i in range(len(ds)):
        d = ds[i]

        y_sim = d['y_sim']
        preds_sim = d['preds_sim']
        y = d['y']
        pred = d['pred']
        df_distribution = pd.DataFrame({'y_sim': y_sim,
                                       'preds_sim': preds_sim,
                                       'scores': expls.data['scores'][i]})

        axes[i, 0].imshow(np.squeeze(d['x']), cmap='gray')

```

(continues on next page)

(continued from previous page)

```

axes[i, 0].axis('off')
if i == 0:
    title_orig = "Original instance"
    axes[i, 0].set_title(f"{title_orig} \n " +
                        f"{len(title_orig) * '='} \n " +
                        f"Label: {d['y']} - Prediction: {d['pred']} ")
else:
    axes[i, 0].set_title(f"Label: {d['y']} - Prediction: {d['pred']}")
df_y = df_distribution.groupby('y_sim')['scores'].mean().sort_
↪ values(ascending=False)
df_y.plot(kind='bar', ax=axes[i, 1])
if i == 0:
    title_true_class = "Averaged scores for each true class in reference set"
    axes[i, 1].set_title(f"{title_true_class} \n " +
                        f"{len(title_true_class) * '='} \n ")
df_preds = df_distribution.groupby('preds_sim')['scores'].mean().sort_
↪ values(ascending=False)
df_preds.plot(kind='bar', ax=axes[i, 2])
if i == 0:
    title_pred_class = "Averaged scores for each predicted class in reference set"
↪ "
    axes[i, 2].set_title(f"{title_pred_class} \n " +
                        f"{len(title_pred_class) * '='} \n ")

plt.show()

def append_int(num):
    """Converts an integer into an ordinal (ex. 1 -> 1st, 2 -> 2nd, etc.).

    Parameters
    -----
    num
        Integer number

    Returns
    -----
    Ordinal suffixes
    """
    if num > 9:
        secondToLastDigit = str(num)[-2]
        if secondToLastDigit == '1':
            return 'th'
    lastDigit = num % 10
    if (lastDigit == 1):
        return 'st'
    elif (lastDigit == 2):
        return 'nd'
    elif (lastDigit == 3):
        return 'rd'
    else:
        return 'th'

```


Load data

Loading and preparing the MNIST data set.

```
[3]: train, test = tf.keras.datasets.mnist.load_data()
X_train, y_train = train
X_test, y_test = test
test_labels = y_test.copy()
train_labels = y_train.copy()

X_train = X_train.reshape(-1, 28, 28, 1).astype('float64') / 255
X_test = X_test.reshape(-1, 28, 28, 1).astype('float64') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)

(60000, 28, 28, 1) (60000, 10) (10000, 28, 28, 1) (10000, 10)
```

Train model

Train a convolutional neural network on the MNIST dataset. The model includes 2 convolutional layers and it reaches a test accuracy of 0.98. If `save_model = True`, a local folder `./model_mnist` will be created and the trained model will be saved in that folder. If the model was previously saved, it can be loaded by setting `load_mnist_model = True`.

```
[4]: load_mnist_model = False
save_model = True

[ ]: filepath = './model_mnist/' # change to directory where model is saved
if load_mnist_model:
    model = tf.keras.models.load_model(os.path.join(filepath, 'model.h5'))
else:
    # define model
    inputs = Input(shape=(X_train.shape[1:]), dtype=tf.float64)
    x = Conv2D(64, 2, padding='same', activation='relu')(inputs)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(.3)(x)

    x = Conv2D(32, 2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=2)(x)
    x = Dropout(.3)(x)

    x = Flatten()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(.5)(x)
    logits = Dense(10, name='logits')(x)
    outputs = Activation('softmax', name='softmax')(logits)
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    # train model
```

(continues on next page)

(continued from previous page)

```

model.fit(X_train,
          y_train,
          epochs=6,
          batch_size=256,
          verbose=1,
          validation_data=(X_test, y_test)
        )
if save_model:
    if not os.path.exists(filepath):
        os.makedirs(filepath)
    model.save(os.path.join(filepath, 'model.h5'))

```

Find similar instances

Initializing a GradientSimilarity explainer instance:

```
[6]: gsm = GradientSimilarity(model, categorical_crossentropy, precompute_grads=True, sim_fn=
    ↪ 'grad_cos')
```

Selecting a reference set of 1000 random samples from the training set. The GradientSimilarity explainer will find the most similar instances among those. This downsampling step is performed to speed up the fit step.

```
[7]: idxs_ref = np.random.choice(len(X_train), 1000, replace=False)
X_ref, y_ref = X_train[idxs_ref], y_train[idxs_ref]
```

Fitting the explainer on the reference data:

```
[8]: gsm.fit(X_ref, y_ref)
[8]: GradientSimilarity(meta={
    'name': 'GradientSimilarity',
    'type': ['whitebox'],
    'explanations': ['local'],
    'params': {
        'sim_fn_name': 'grad_cos',
        'store_grads': True,
        'backend_name': 'tensorflow',
        'task_name': 'classification'
    },
    'version': '0.6.6dev'
})
```

Selecting 5 random instances from the test set:

```
[9]: idxs_samples = np.random.choice(len(X_test), 5, replace=False)
X_sample, y_sample = X_test[idxs_samples], y_test[idxs_samples]
preds = model(X_sample).numpy().argmax(axis=1)
```

Getting the most similar instances for the each of the 5 test samples:

```
[10]: expls = gsm.explain(X_sample, y_sample)
```

Visualizations

Building a dictionary for each sample for visualization purposes. Each dictionary contains

- The original image `x`.
- The corresponding label `y`.
- The corresponding model's prediction `pred`.
- The corresponding reference labels ordered by similarity `y_sim`.
- The corresponding model's predictions for the reference set `preds_sim`.

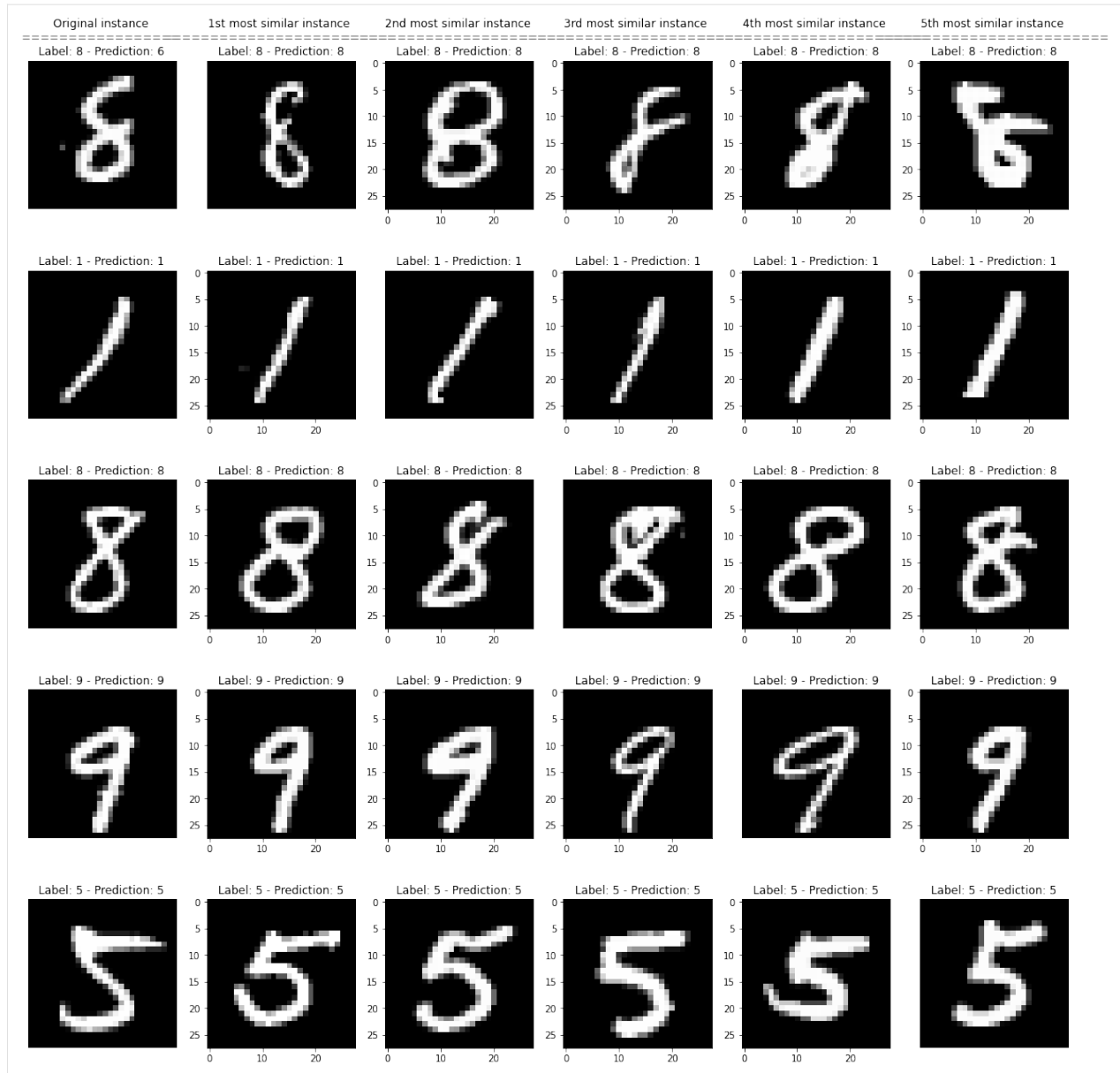
```
[11]: ds = []
      for j in range(len(X_sample)):
          y_sim = y_ref[expls.data['ordered_indices'][j]].argmax(axis=1)
          X_sim = X_ref[expls.data['ordered_indices'][j]]
          preds_sim = model(X_sim).numpy().argmax(axis=1)

          d = {'x': X_sample[j],
              'y': y_sample[j].argmax(),
              'pred': preds[j],
              'y_sim': y_sim,
              'preds_sim': preds_sim}
          ds.append(d)
```

Showing the 5 most similar instances for each of the test instances, ordered from the most similar to the least similar.

Most similar instances

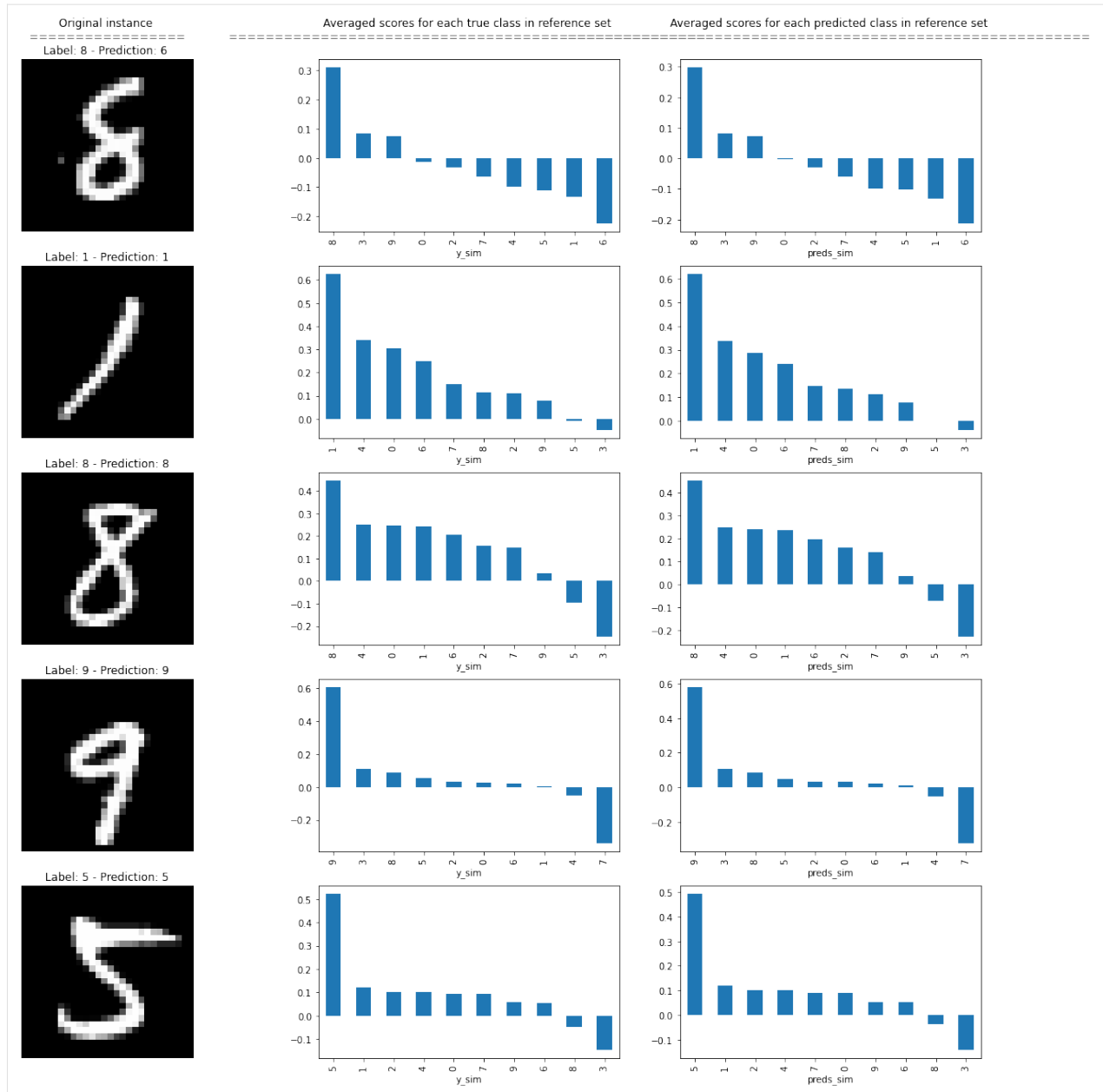
```
[12]: plot_similar(ds, expls)
```



Most similar labels distributions

Showing the average similarity scores for each group of instances in the reference set belonging to the same true class and to same predicted class.

```
[13]: plot_distributions(ds, expls)
```



The plots show how the instances belonging to the same class (and the instances classified by the model as belonging to the same class) of the instance of interest have on average higher similarity scores, as expected.

8.14 Tree SHAP

8.14.1 Explaining Tree Models with Interventional Feature Perturbation Tree SHAP

Note

To enable SHAP support, you may need to run

```
pip install alibi[shap]
```

```
[ ]: # shap.summary_plot currently doesn't work with matplotlib>=3.6.0,  
# see bug report: https://github.com/slundberg/shap/issues/2687  
!pip install matplotlib==3.5.3
```

Introduction

This example shows how to apply interventional Tree SHAP to compute shap values exactly for an `xgboost` model fitted to the `Adult` dataset (binary classification task). Furthermore, the shap values computed by Kernel SHAP, an approximate feature attribution method, are shown to converge to the interventional Tree SHAP contributions given a sufficiently large number of model evaluations.

This example will use the `xgboost` library (v1.6.1). The latest version can be installed with:

```
[1]: !pip install -q xgboost
```

```
[2]: import json  
import pickle  
import shap  
shap.initjs()  
  
import numpy as np  
import matplotlib.pyplot as plt  
import xgboost as xgb  
  
from alibi.datasets import fetch_adult  
from alibi.explainers import KernelShap, TreeShap  
from collections import defaultdict, Counter  
from functools import partial  
from itertools import product, zip_longest  
  
from scipy.special import expit  
invlogit=expit  
from sklearn.metrics import accuracy_score, confusion_matrix  
from sklearn.utils import resample  
  
from timeit import default_timer as timer  
  
<IPython.core.display.HTML object>
```

Data preparation

Load and split

The `fetch_adult` function returns a Bunch object containing features, targets, feature names and a mapping of categorical variables to numbers.

```
[3]: adult = fetch_adult()
      adult.keys()

[3]: dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])
```

```
[4]: data = adult.data
      target = adult.target
      target_names = adult.target_names
      feature_names = adult.feature_names
      category_map = adult.category_map
```

Note that for your own datasets you can use the utility function `gen_category_map` imported from `alibi.utils` to create the category map.

```
[5]: np.random.seed(0)
      data_perm = np.random.permutation(np.c_[data, target])
      data = data_perm[:, :-1]
      target = data_perm[:, -1]
```

```
[6]: idx = 30000
      X_train, y_train = data[:idx, :], target[:idx]
      X_test, y_test = data[idx+1:, :], target[idx+1:]
```

xgboost wraps arrays using `DMatrix` objects, optimised for both memory efficiency and training speed.

```
[7]: def wrap(arr):
      return np.ascontiguousarray(arr)

      dtrain = xgb.DMatrix(
          wrap(X_train),
          label=wrap(y_train),
          feature_names=feature_names,
      )

      dtest = xgb.DMatrix(wrap(X_test), label=wrap(y_test), feature_names=feature_names)
```

Finally, a matrix that contains the raw string values for categorical variables (used for display) is created:

```
[8]: def _decode_data(X, feature_names, category_map):
      """
      Given an encoded data matrix `X` returns a matrix where the
      categorical levels have been replaced by human readable categories.
      """

      X_new = np.zeros(X.shape, dtype=object)
      for idx, name in enumerate(feature_names):
          categories = category_map.get(idx, None)
```

(continues on next page)

(continued from previous page)

```

    if categories:
        for j, category in enumerate(categories):
            encoded_vals = X[:, idx] == j
            X_new[encoded_vals, idx] = category
    else:
        X_new[:, idx] = X[:, idx]

    return X_new

decode_data = partial(_decode_data,
                      feature_names=feature_names,
                      category_map=category_map)

```

```
[9]: X_display = decode_data(X_test)
```

```
[10]: X_display
```

```

[10]: array([[52, 'Private', 'Associates', ..., 0, 60, 'United-States'],
          [21, 'Private', 'High School grad', ..., 0, 20, 'United-States'],
          [43, 'Private', 'Dropout', ..., 0, 50, 'United-States'],
          ...,
          [23, 'Private', 'High School grad', ..., 0, 40, 'United-States'],
          [45, 'Local-gov', 'Doctorate', ..., 0, 45, 'United-States'],
          [25, 'Private', 'High School grad', ..., 0, 48, 'United-States']],
      dtype=object)

```

Model definition

The model fitted in the xgboost fitting example will be explained. The confusion matrix of this model is shown below.

```

[11]: def plot_conf_matrix(y_test, y_pred, class_names):
    """
    Plots confusion matrix. Taken from:
    http://queirozfb.com/entries/visualizing-machine-learning-models-examples-with-scikit-learn-and-matplotlib
    """

    matrix = confusion_matrix(y_test, y_pred)

    # place labels at the top
    plt.gca().xaxis.tick_top()
    plt.gca().xaxis.set_label_position('top')

    # plot the matrix per se
    plt.imshow(matrix, interpolation='nearest', cmap=plt.cm.Blues)

    # plot colorbar to the right
    plt.colorbar()

    fmt = 'd'

```

(continues on next page)

(continued from previous page)

```

# write the number of predictions in each bucket
thresh = matrix.max() / 2.
for i, j in product(range(matrix.shape[0]), range(matrix.shape[1])):

    # if background is dark, use a white number, and vice-versa
    plt.text(j, i, format(matrix[i, j], fmt),
             horizontalalignment="center",
             color="white" if matrix[i, j] > thresh else "black")

tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names, rotation=45)
plt.yticks(tick_marks, class_names)
plt.tight_layout()
plt.ylabel('True label',size=14)
plt.xlabel('Predicted label',size=14)
plt.show()

def predict(xgb_model, dataset, proba=False, threshold=0.5):
    """
    Predicts labels given a xgboost model that outputs raw logits.
    """

    y_pred = model.predict(dataset) # raw logits are predicted
    y_pred_proba = invlogit(y_pred)
    if proba:
        return y_pred_proba
    y_pred_class = np.zeros_like(y_pred)
    y_pred_class[y_pred_proba >= threshold] = 1 # assign a label

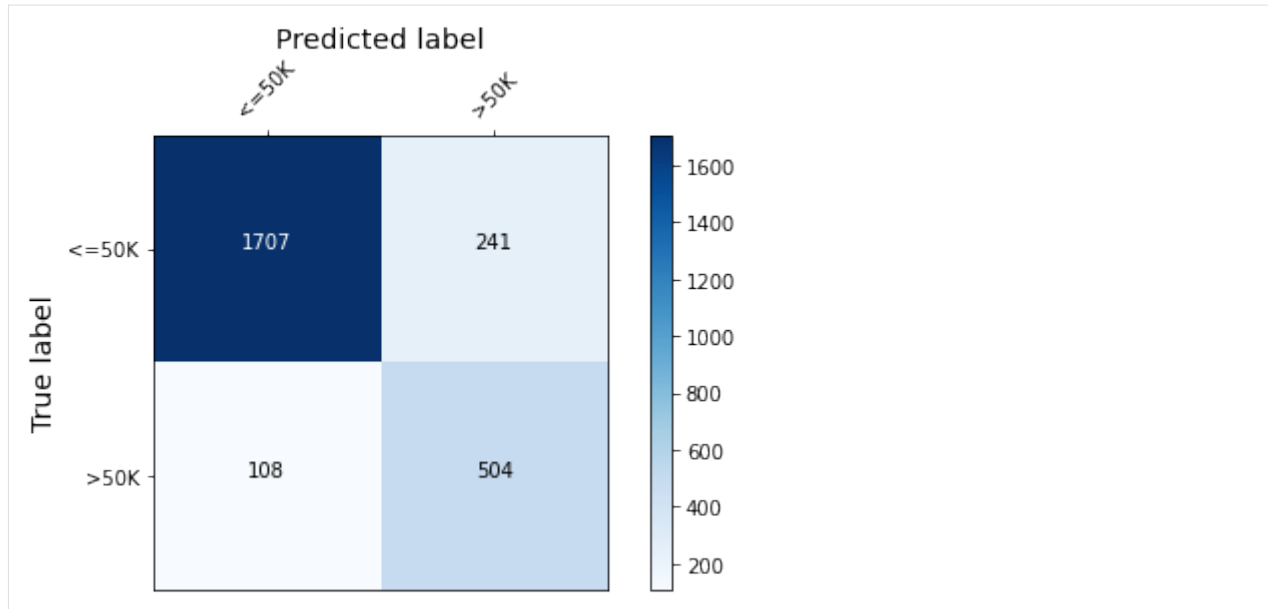
    return y_pred_class

```

```
[12]: model = xgb.Booster()
      model.load_model('assets/adult_xgb.mdl')
```

```
[13]: y_pred_train = predict(model, dtrain)
      y_pred_test = predict(model, dtest)
```

```
[14]: plot_conf_matrix(y_test, y_pred_test, target_names)
```



```
[15]: print(f'Train accuracy: {round(100*accuracy_score(y_train, y_pred_train), 4)} %.')
      print(f'Test accuracy: {round(100*accuracy_score(y_test, y_pred_test), 4)} %.')
```

```
Train accuracy: 87.75 %.
Test accuracy: 86.3672%.
```

Explaining xgboost with interventional Tree SHAP: global knowledge from local explanations

Recall that the goal of shap values computation for an instance x is to attribute the difference $f(x) - \mathbb{E}_{\mathcal{D}}[f(x)]$ to M input features. Here \mathcal{D} represents the background data. Unlike the *path-dependent perturbation* algorithm which exploits the tree structure and cover information (derived from the training data) to obviate the need for a background dataset, the interventional perturbation algorithm follows a similar idea to *Kernel SHAP* and uses a background dataset to compute the expected value as the average of the leaves where the background samples fall plus the baseline model offset (1). As explained in the algorithm *overview*, this allows explaining nonlinear transformations of the model output, so this method can be used to explain loss function fluctuations.

As discussed in [1] and detailed in the *overview*, this perturbation method enforces the conditional independence $x_S \perp x_{\bar{S}}$ where \bar{S} is a subset of missing features. This section shows that this method is consistent with the path-dependent perturbation method, in the sense that it leads to very similar analysis conclusions assuming an appropriate background dataset is used.

Because the background dataset contains 30,000 examples, the next part of the example would be in principle a **long running**. In practice, sufficient accuracy can be achieved using a couple of hundred samples (the library authors recommend anywhere between 100 and 1000 examples), provided that the samples chosen represent the underlying distribution accurately (i.e., they cover the entire support of the distribution). You can skip the computation by setting `COMPUTE_SHAP = False` and can **load the results** by calling the `load_shap_values` function.

Warning

The upstream implementation of interventional TreeShap supports only up to 100 samples in the background dataset. A larger background dataset will be sampled with replacement to 100 instances. Some of the issues related to this limitation have been reported [here](#) and [here](#). Thus, we will use only 100 background samples for the following experiments.

```
[16]: background_indices = np.random.choice(len(X_train), size=100, replace=False)
background_data = X_train[background_indices]

tree_explainer_interventional = TreeShap(model, model_output='raw', task='classification
↳')
tree_explainer_interventional.fit(background_data=background_data)

COMPUTE_SHAP = False # whether to compute the SHAP values from scratch (the computation
↳is quite fast).

if COMPUTE_SHAP:
    explanation = tree_explainer_interventional.explain(X_test)

    with open('assets/shap_interv.pkl', 'wb') as f:
        pickle.dump(explanation.shap_values[0], f)
```

Predictor returned a scalar value. Ensure the output represents a probability or
↳decision score as opposed to a classification label!

```
[17]: def load_shap_values():
    with open('assets/shap_interv.pkl', 'rb') as f:
        shap_interventional = pickle.load(f)

    return shap_interventional
```

```
[18]: interventional_shap_values = load_shap_values()
```

Note that the local accuracy property holds for all examples.

```
[19]: errs = np.abs(model.predict(dtest) - tree_explainer_interventional.expected_value -
↳interventional_shap_values.sum(1))
print(Counter(np.round(errs, 2)))

Counter({0.0: 2560})
```

```
[20]: shap.summary_plot(interventional_shap_values, X_test, feature_names)
```

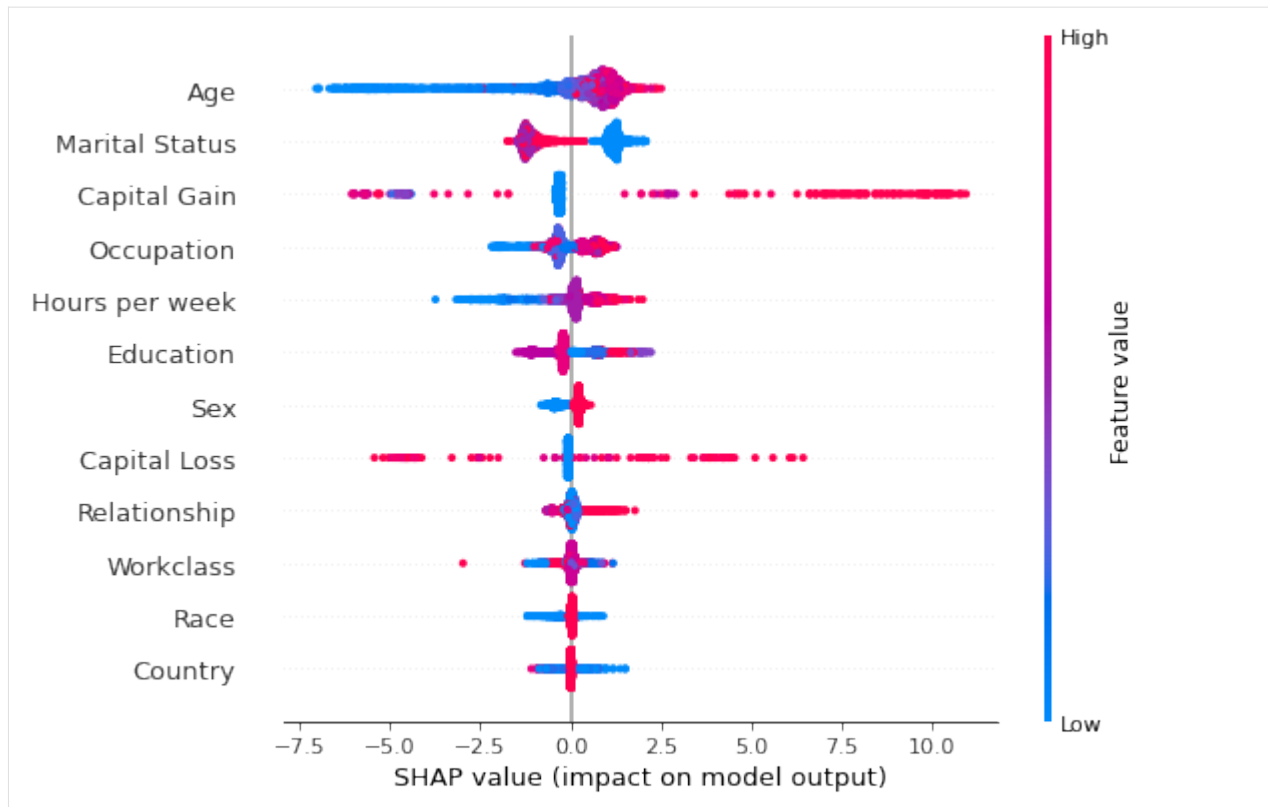


Figure 1: Summary plot of the interventional perturbation Tree SHAP explanations for the test set

```
[21]: shap.summary_plot(interventional_shap_values, X_test, feature_names, plot_type='bar')
```

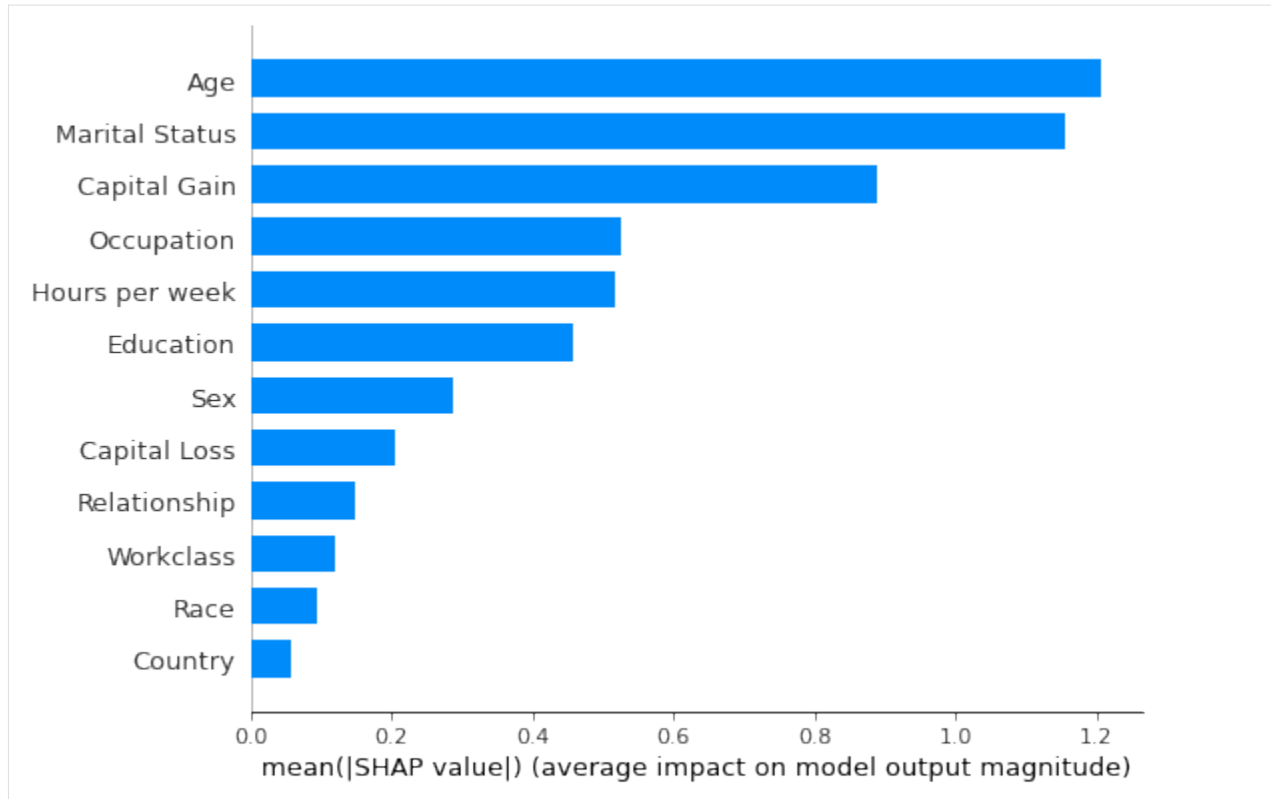


Figure 2: Most important features as predicted by the interventional perturbation Tree SHAP algorithm

One might be tempted to proceed to compare the feature rankings displayed above with the ranking provided by the path-dependent Tree SHAP [example](#). However, these algorithms have different ways of estimating the effect of missing features and:

1. The length of the bar represents the average magnitude of the points in the summary plot above; each point is the average of the shap values computed for a given instance x with respect to R different background samples. Hence, one can consider that for each instance to be explained the shap value of the j th feature is a random variable, denoted by $\Phi_{i,j}$. One way to define the importance of the j th feature, I_j , is

$$I_j = \frac{1}{N} \sum_{i=1}^N |\mathbb{E}[\Phi_{i,j}]|,$$

where the expectation is taken over the background distribution and N is the number of instances explained. This corresponds to the notion of feature importance according to which a feature is important for explaining the model behaviour over a given dataset if:

- either the instances to explained are consistently affected by the feature, or the feature has a particularly large impact for certain subgroups and a small or moderate impact for the remainder. Traditional global explanation feature importances hide this information whereas the summary plot reveals why a particular feature was deemed important
- locally, one also requires that cancellation effects are not significant. In other words, for a particular instance, a feature would be considered as not important if, across different backgrounds, cancellation effects result in a small average for the effect.

It should be noted that the error I_j is inversely proportional to the square root of the size of the background dataset for a given dataset to be explained, so it is important to select a sufficient number of background samples in order to reduce the error of this estimate.

2. The two methods explain the dataset with respect to different expected values, so the contributions will be different. This also arises because of the different set of conditional assumptions are made when estimating the individual contributions, as explained in the algorithm [overview](#).

Instead of analysing feature importance rankings, it is perhaps more instructive to look at the dependence plots and see if the conclusions from the previous model interpretation hold. Although the decision plots in Figure 3 show the same patterns as their counterparts in the path-dependent [example](#), different variables are found to have the strongest interaction with the variables of interest so the colouring of the plot is different. This is expected since the different conditional independence assumptions give rise to different magnitudes for the shap values, and therefore the estimations for the Pearson coefficients will be affected.

```
[22]: def _dependence_plot(features, shap_values, dataset, feature_names, category_map,
    ↪ display_features=None, **kwargs):
    """
    Plots dependence plots of specified features in a grid.

    features: List[str], List[Tuple[str, str]]
        Names of features to be plotted. If List[str], then shap
        values are plotted as a function of feature value, coloured
        by the value of the feature determined to have the strongest
        interaction (empirically). If List[Tuple[str, str]], shap
        interaction values are plotted.
    display_features: np.ndarray, N x F
        Same as dataset, but contains human readable values
        for categorical levels as opposed to numerical values
    """

    def _set_fonts(fig, ax, fonts=None, set_cbar=False):
        """
        Sets fonts for axis labels and colobar.
        """

        ax.xaxis.label.set_size(xlabelfontsize)
        ax.yaxis.label.set_size(ylabelfontsize)
        ax.tick_params(axis='x', labelsize=xtickfontsize)
        ax.tick_params(axis='y', labelsize=ytickfontsize)
        if set_cbar:
            fig.axes[-1].tick_params(labelsize=cbartickfontsize)
            fig.axes[-1].tick_params(labelrotation=cbartickrotation)
            fig.axes[-1].yaxis.label.set_size(cbarlabelfontsize)

    # parse plotting args
    figsize = kwargs.get("figsize", (15, 10))
    nrows = kwargs.get('nrows', len(features))
    ncols = kwargs.get('ncols', 1)
    xlabelfontsize = kwargs.get('xlabelfontsize', 14)
    xtickfontsize = kwargs.get('xtickfontsize', 11)
    ylabelfontsize = kwargs.get('ylabelfontsize', 14)
    ytickfontsize = kwargs.get('ytickfontsize', 11)
    cbartickfontsize = kwargs.get('cbartickfontsize', 14)
    cbartickrotation = kwargs.get('cbartickrotation', 10)
    cbarlabelfontsize = kwargs.get('cbarlabelfontsize', 14)
    rotation_orig = kwargs.get('xticklabelrotation', 25)
```

(continues on next page)

(continued from previous page)

```

alpha = kwargs.get("alpha", 1)
x_jitter_orig = kwargs.get("x_jitter", 0.8)
grouped_features = list(zip_longest(*[iter(features)] * ncols))

fig, axes = plt.subplots(nrows, ncols, figsize=figsize)
if nrows == len(features):
    axes = list(zip_longest(*[iter(axes)] * 1))

for i, (row, group) in enumerate(zip(axes, grouped_features), start=1):
    # plot each feature or interaction in a subplot
    for ax, feature in zip(row, group):
        # set x-axis ticks and labels and x-jitter for categorical variables
        if not feature:
            continue
        if isinstance(feature, list) or isinstance(feature, tuple):
            feature_index = feature_names.index(feature[0])
        else:
            feature_index = feature_names.index(feature)
        if feature_index in category_map:
            ax.set_xticks(np.arange(len(category_map[feature_index])))
            if i == nrows:
                rotation = 90
            else:
                rotation = rotation_orig
            ax.set_xticklabels(category_map[feature_index], rotation=rotation,
↪ fontsize=22)
            x_jitter = x_jitter_orig
        else:
            x_jitter = 0

        shap.dependence_plot(feature,
                             shap_values,
                             dataset,
                             feature_names=feature_names,
                             display_features=display_features,
                             interaction_index='auto',
                             ax=ax,
                             show=False,
                             x_jitter=x_jitter,
                             alpha=alpha
                             )

    if i != nrows:
        ax.tick_params('x', labelrotation=rotation_orig)
    _set_fonts(fig, ax, set_cbar=True)

plot_dependence = partial(
    _dependence_plot,
    feature_names=feature_names,
    category_map=category_map,
)

```

Warning

For the following plots to run the matplotlib version needs to be `<3.5.0`. This is because of an upstream issue of how the `shap.dependence_plot` function is handled in the `shap` library. An issue tracking it can be found [here](#).

```
[23]: plot_dependence(['Marital Status', 'Age', 'Hours per week', 'Occupation'],
                    interventional_shap_values,
                    X_test,
                    display_features=X_display,
                    nrows=2,
                    ncols=2,
                    figsize=(22, 10),
                    alpha=0.5)
```

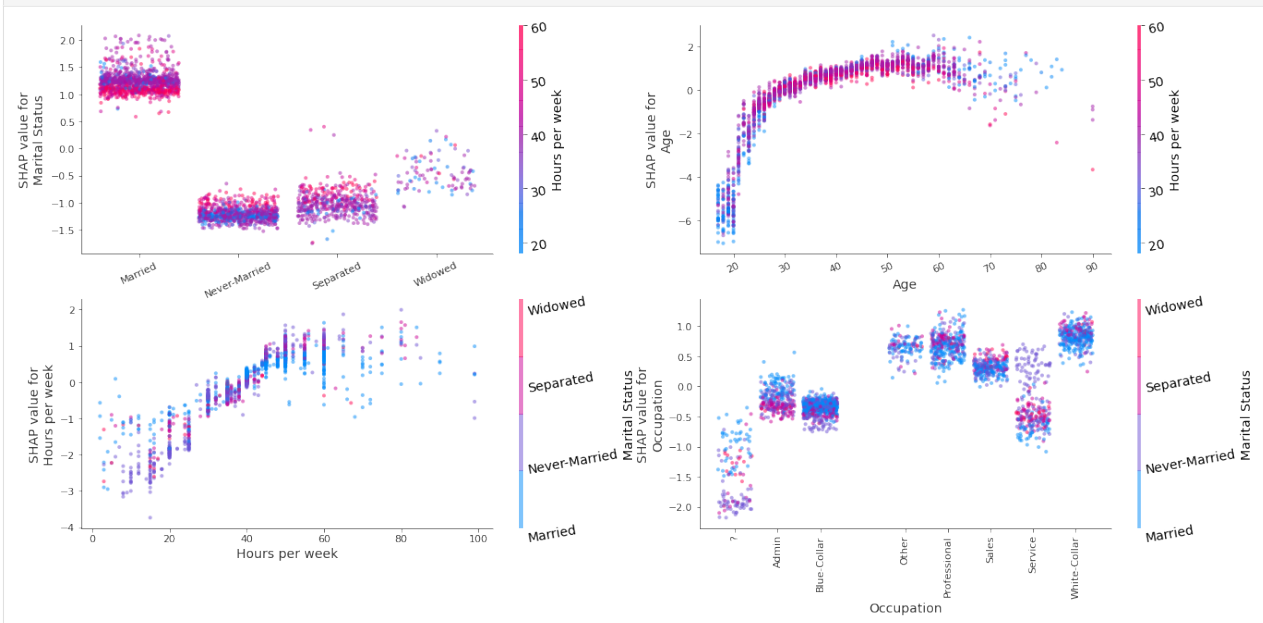


Figure 3: Decision plots of the variables Marital Status, Age, Sex, Race, Occupation, Education using the interventional perturbation Tree SHAP algorithm for the test set

By changing, value of feature below, one can recolour the decision plots according to the interactions estimate from the path-dependent perturbation example. Generally, the same interaction patterns are observed, with the exception of Age, where the interaction with the Capital Gain feature is not conclusive.

```
[24]: path_dep_interactions = {
    'Marital Status': 'Hours per week',
    'Age': 'Capital Gain',
    'Hours per week': 'Age',
    'Occupation': 'Sex',
}
```

```
[25]: feature = 'Occupation'
x_jitter = 0.5 if feature in ['Occupation', 'Marital Status'] else 0
shap.dependence_plot(feature,
                    interventional_shap_values,
                    X_test,
```

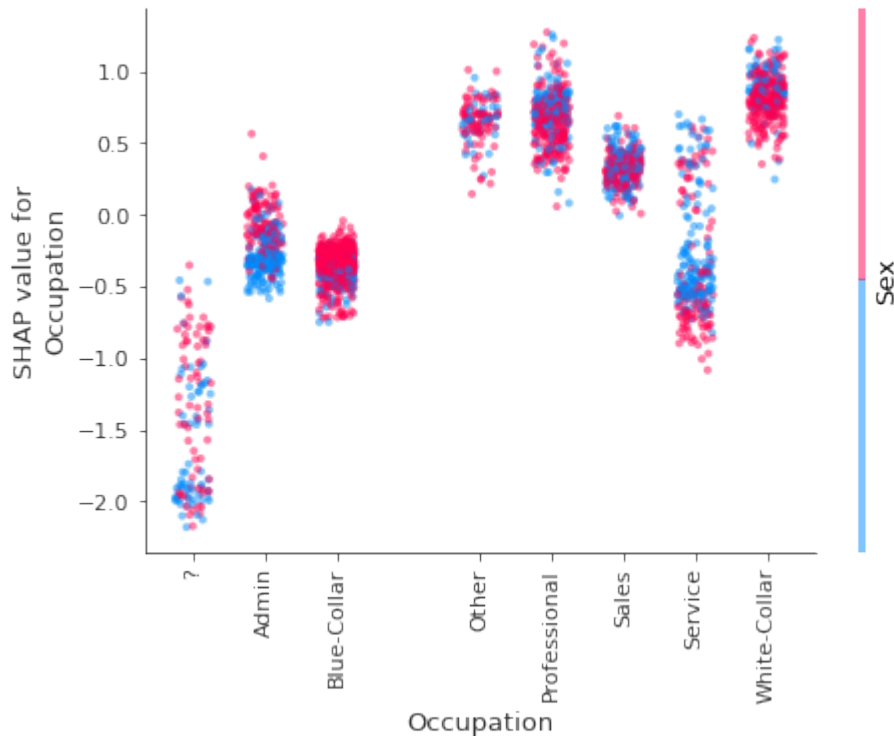
(continues on next page)

(continued from previous page)

```

feature_names=feature_names,
display_features=X_display,
interaction_index=path_dep_interactions[feature],
alpha=0.5,
x_jitter=x_jitter
)

```



If interaction effects are of interest, these can be computed exactly using the path-dependent perturbation algorithm as opposed to approximated.

White-box vs black-box model explanations: a comparison with Kernel SHAP

The main drawback of model-agnostic methods such as [Kernel SHAP](#) is their sample complexity, which leads to variability in the results obtained. Given enough samples, the feature attributions estimated Kernel SHAP algorithm approach their exact values and give rise to the same feature importance rankings, as shown below.

Below, both the Tree SHAP and Kernel SHAP algorithms are used to explain 100 instances from the test set using a background dataset of 100 samples (recall the background dataset size limitation of interventional TreeShap). For the Kernel SHAP algorithm, each explanation is computed 10 times to account for the variability in the estimation.

```

[26]: n_background_samples = 100 # same background dataset size limitation of interventional_
      ↪ TreeShap
      n_explained = 100
      background_dataset, y_background = resample(X_train, y_train, n_samples=n_background_
      ↪ samples, replace=False, random_state=0)

```

```
[27]: X_display_background = decode_data(background_dataset)
X_explain = X_test[:n_explained, :]
```

```
[28]: tree_explainer = TreeShap(model, model_output='raw', task='classification')
tree_explainer.fit(background_dataset)
explanation = tree_explainer.explain(X_explain)
tree_shap_values = explanation.shap_values[0]
```

Predictor returned a scalar value. Ensure the output represents a probability or
 ↳ decision score as opposed to a classification label!

xgboost requires the model inputs to be a DMatrix instance, so predict_fcn needs to account for this transformation to avoid errors.

```
[29]: predict_fcn = lambda x: model.predict(xgb.DMatrix(x, feature_names=feature_names))
kernel_explainer = KernelShap(predict_fcn)
```

```
[30]: kernel_explainer.fit(background_dataset)
```

Predictor returned a scalar value. Ensure the output represents a probability or
 ↳ decision score as opposed to a classification label!

```
[30]: KernelShap(meta={
    'name': 'KernelShap',
    'type': ['blackbox'],
    'task': 'classification',
    'explanations': ['local', 'global'],
    'params': {
        'link': 'identity',
        'group_names': None,
        'grouped': False,
        'groups': None,
        'weights': None,
        'summarise_background': False,
        'summarise_result': None,
        'transpose': False,
        'kwargs': {}
    },
    'version': '0.7.1dev'
})
```

To assess convergence, Kernel SHAP is run with the numbers of samples specified in `n_samples` for `n_runs`. Since the computation is quite slow, you can skip the computation and can **load the precomputed results** by setting `COMPUTE_SHAP = False`.

```
[31]: n_runs = 5
# There is no point going beyond 2^(num_features) = 2^12 since larger number will be
↳ truncated to 2^(num_features).
# 2^(num_features) represents the maximum number of enumerable subsets of the set of
↳ input features.
n_samples = [64, 128, 512, 1024, 4096]
```

```
[32]: COMPUTE_SHAP = False # whether to compute the SHAP values from scratch (the computation
↳ is quite slow).
```

(continues on next page)

(continued from previous page)

```

if COMPUTE_SHAP:
    results = defaultdict(list)
    times = defaultdict(list)

    for n_samp in n_samples:
        print(f"Number of samples {n_samp}")
        for run in range(n_runs):
            t_start = timer()
            exp = kernel_explainer.explain(X_explain, nsamples=n_samp, l1_reg=False)
            t_end = timer()
            times[str(n_samp)].append(t_end - t_start)
            results[str(n_samp)].append(exp.shap_values[0])

        results['time'] = times

    with open('assets/kernel_convergence.pkl', 'wb') as f:
        pickle.dump(results, f)

```

```

[33]: with open('assets/kernel_convergence.pkl', 'rb') as f:
        convergence_data = pickle.load(f)

```

To compare the two algorithms, the mean absolute deviation from the ground truth provided by the Tree SHAP algorithm with interventional feature perturbation is computed. For each number of samples, either the maximum mean absolute deviation across the feature, or the mean of this quantity across the features is computed. This calculation can be performed for one instance, or averaged across an entire distribution. The plots below show that all these quantities approach to the ground truth values. A threshold of 1% from the effect of the most important feature (Marital Status) is depicted.

```

[34]: def get_errors(tree_shap_values, convergence_data, instance_idx=None):
    """
    Compute the mean and max maximum absolute deviation of Kernel SHAP values
    from Tree SHAP values for a specific instance or as an average over instances.
    If instance_idx is set, then the errors are computed at instance level.
    """

    mad = []
    for key in convergence_data:
        if key != 'time':
            mad.append(np.abs(tree_shap_values - np.mean(convergence_data[key], axis=0)))

    if instance_idx is not None:
        err_max = [max(x[instance_idx, :]) for x in mad]
        err_mean = [np.mean(x[instance_idx, :]).item() for x in mad]
    else:
        err_max = [max(x.mean(axis=0)) for x in mad]
        err_mean = [np.mean(x.mean(axis=0)).item() for x in mad]

    return err_max, err_mean

def plot_convergence(err_mean, err_max, n_samples, threshold, instance_idx=None):

```

(continues on next page)

(continued from previous page)

```

"""
Plots the average error across the features and the maximum error across
features as a function of the number of samples Kernel SHAP uses to estimate
the contributions.
"""

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.loglog(n_samples, err_max, '--*')
ax1.plot([0] + n_samples, [threshold]*(len(n_samples)+1), '--', color='gray',
↪ linewidth='3')
ax1.grid(True)
ax1.set_ylabel('Estimation error (max over all features)')
ax1.set_xlabel('Number of samples')

ax2.loglog(n_samples, err_mean, '--*')
ax2.plot([0] + n_samples, [threshold]*(len(n_samples)+1), '--', color='gray',
↪ linewidth='3')
ax2.grid(True)
ax2.set_ylabel('Estimation error (mean over all features)')
ax2.set_xlabel('Number of samples')
if instance_idx is not None:
    plt.suptitle(f'Convergence of the Kernel SHAP algorithm to exact shap values.
↪ (instance {instance_idx})')
else:
    plt.suptitle('Convergence of the Kernel SHAP algorithm to exact shap values.
↪ (mean)')

```

```
[35]: threshold = 0.01 * np.max(np.mean(np.abs(tree_shap_values), axis=0))
```

```
[36]: err_max, err_mean = get_errors(tree_shap_values, convergence_data, instance_idx=0)
plot_convergence(err_max, err_mean, n_samples, threshold, instance_idx=0)
```

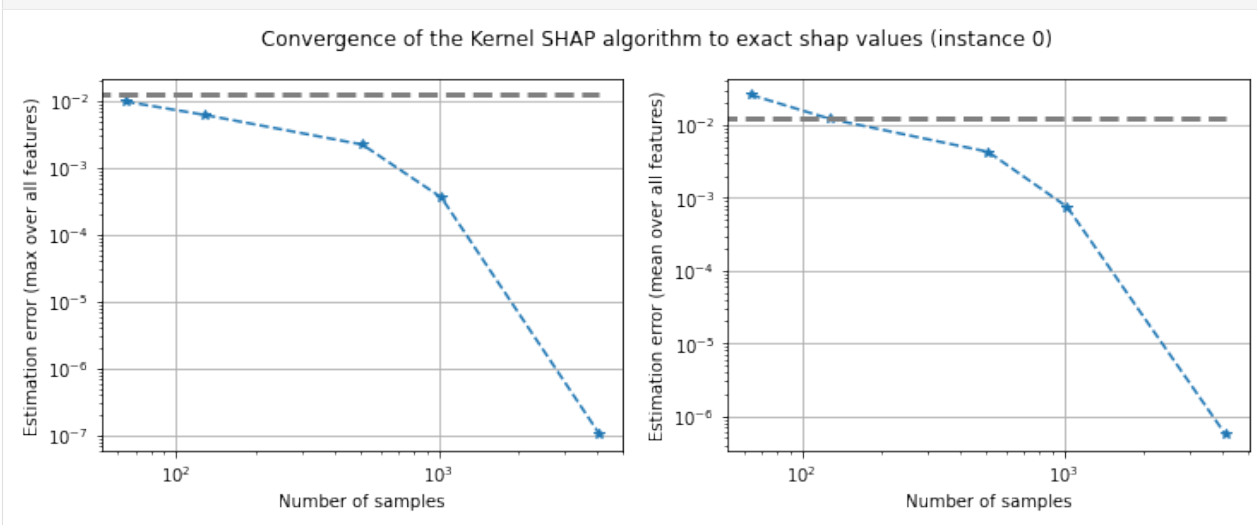


Figure 4: Converge of Kernel SHAP to true values according to the maximum error (left) and mean error (right) for instance 0

```
[37]: e_err_max, e_err_mean = get_errors(tree_shap_values, convergence_data)
      plot_convergence(e_err_max, e_err_mean, n_samples, threshold)
```

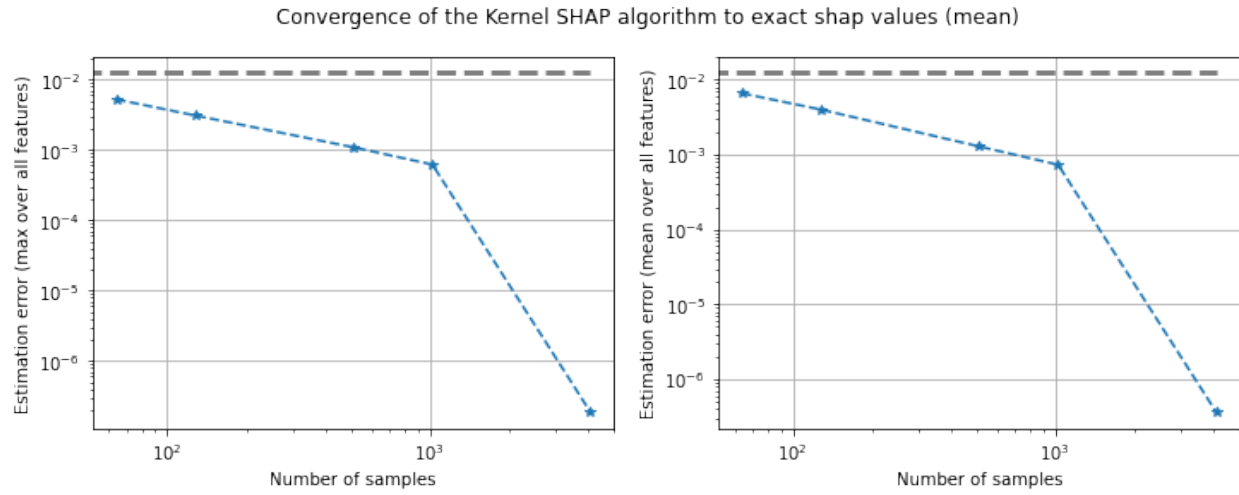


Figure 5: Converge of Kernel SHAP according to the maximum error (left) and mean error (right) averaged across 100 instances

If a high enough number of samples is selected, the algorithms yield the same global patterns, as shown below.

```
[38]: n_explained = 500
      X_explained = X_test[:n_explained, :]
      explanation_500_kernel = kernel_explainer.explain(X_explained, nsamples=1024, l1_
      ↪ reg=False)
      shap_values_500_kernel = explanation_500_kernel.shap_values[0]

      0%|          | 0/500 [00:00<?, ?it/s]
```

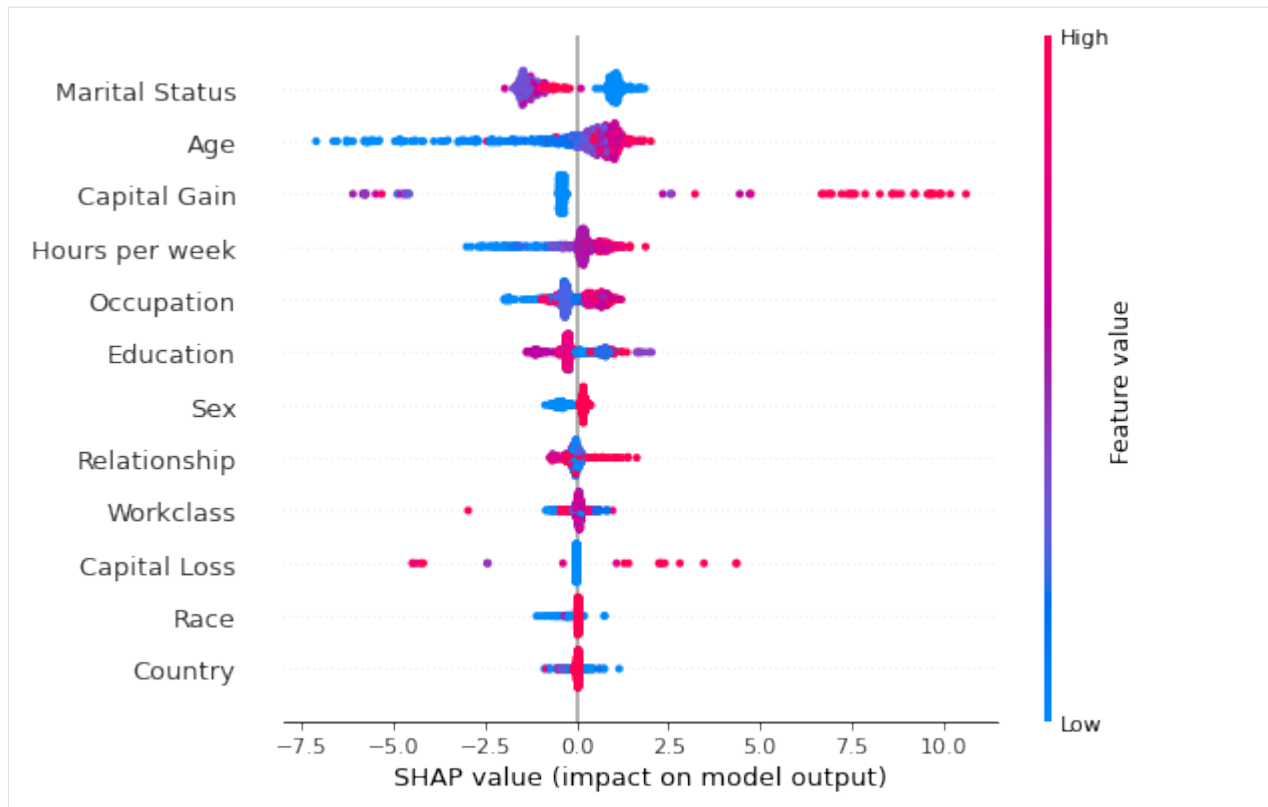
```
[39]: explanation_500_tree = tree_explainer.explain(X_explained)
      shap_values_500_tree = explanation_500_tree.shap_values[0]
```

Again, we can observe that the local accuracy holds.

```
[40]: errs = np.round(np.abs(model.predict(xgb.DMatrix(X_explained, feature_names=feature_
      ↪ names)) - tree_explainer.expected_value - shap_values_500_tree.sum(1)), 2)
      print(Counter(errs))

      Counter({0.0: 500})
```

```
[41]: shap.summary_plot(shap_values_500_tree, X_explained, feature_names)
```



While the Tree SHAP values take a few seconds to compute, the Kernel SHAP takes a few minutes to provide estimates for the shap values. Note that this is also a consequence of the fact that the implementation of Tree SHAP is distributed.

```
[42]: shap.summary_plot(shap_values_500_tree, X_explained, feature_names, plot_type='bar')
```

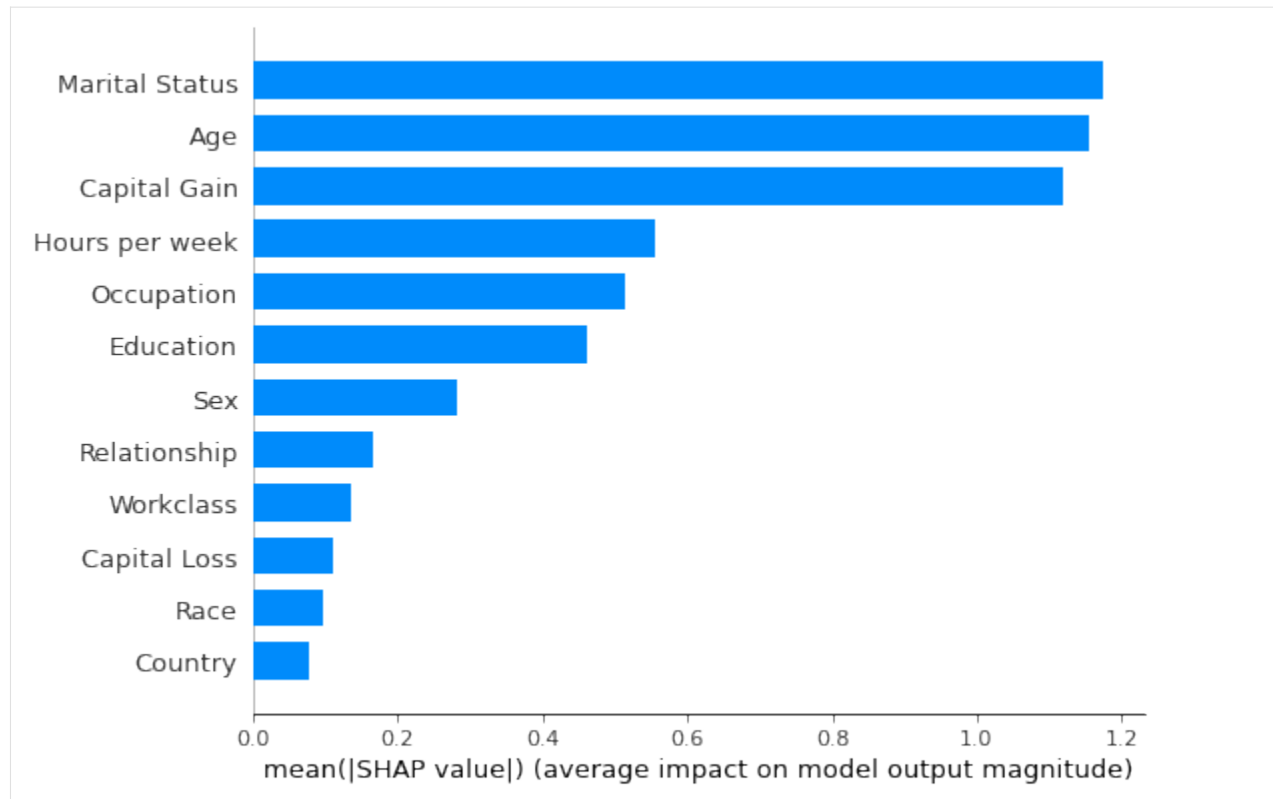


Figure 6: Feature importances estimated using the interventional feature perturbation Tree SHAP algorithm

```
[43]: shap.summary_plot(shap_values_500_kernel, X_explained, feature_names, plot_type='bar')
```

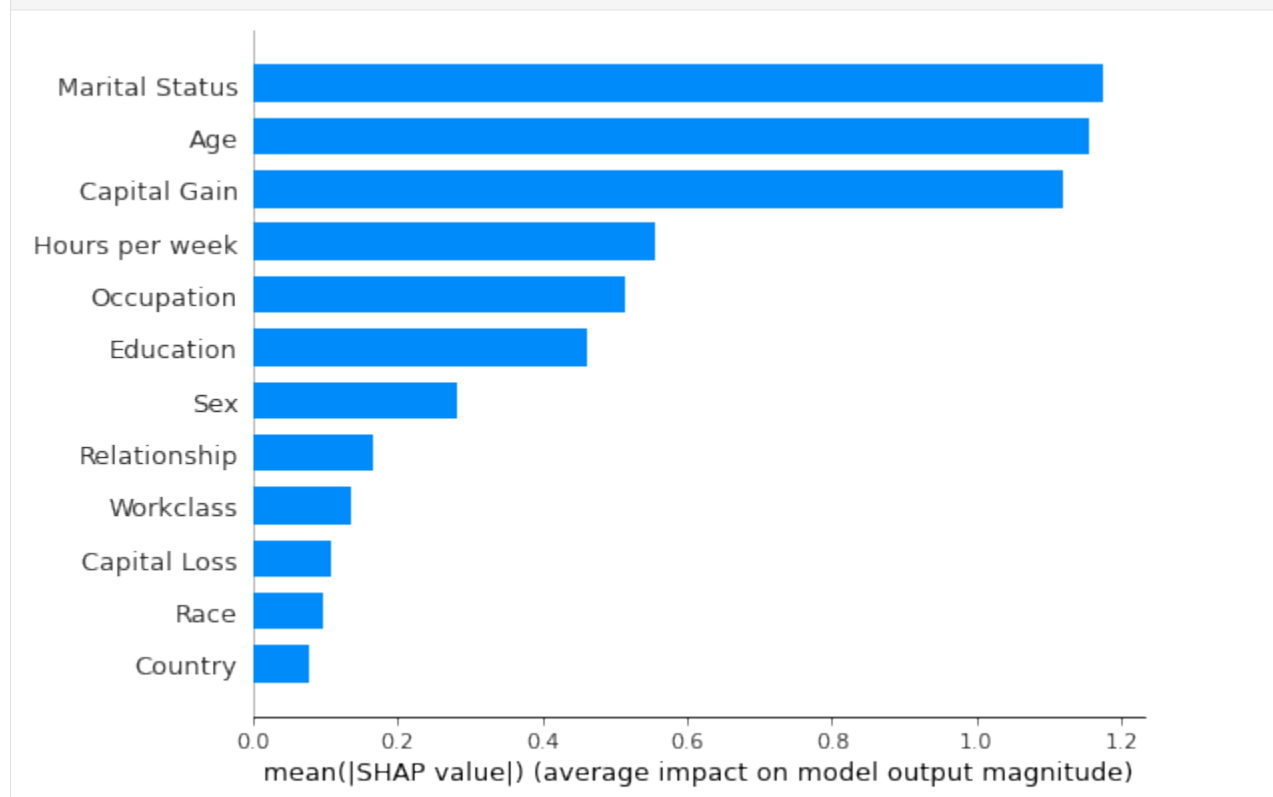


Figure 7: Feature importances estimated using the Kernel SHAP algorithm

```
[44]: print(f"Max absolute deviation from ground truth: {np.round(np.max(np.abs(shap_values_
↪ 500_tree - shap_values_500_kernel)), 4)}.")
print(f"Min absolute deviation from ground truth: {np.round(np.min(np.abs(shap_values_
↪ 500_tree - shap_values_500_kernel)), 4)}.")
```

```
Max absolute deviation from ground truth: 0.0443.
Min absolute deviation from ground truth: 0.0.
```

Since the errors incurred in estimating the shap values are relatively small, the feature importance rankings shown in Figures 6 and 7 are identical.

```
[45]: average_prediction = model.predict(xgb.DMatrix(background_dataset, feature_names=feature_
↪ names)).mean()
kernel_exp_value = kernel_explainer.expected_value
tree_exp_value = tree_explainer.expected_value
print(f"Average prediction on background data is the expected value of kernel explainer:
↪ {np.abs(average_prediction - kernel_exp_value) < 1e-3}")
print(f"Average expected value for kernel explainer is the same as the tree explainer:
↪ {np.abs(kernel_exp_value - tree_exp_value) < 1e-3}")
```

```
Average prediction on background data is the expected value of kernel explainer: True
Average expected value for kernel explainer is the same as the tree explainer: True
```

The expected values of the two explainers are approximately the same.

```
[46]: print(f"The difference between the expected values is {np.round(np.abs(kernel_exp_value -
↪ tree_exp_value), 2)}.")
```

```
The difference between the expected values is 0.0.
```

References

[1] Lundberg, S.M., Erion, G., Chen, H., DeGrave, A., Prutkin, J.M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N. and Lee, S.I., 2020. From local explanations to global understanding with explainable AI for trees. *Nature machine intelligence*, 2(1), pp.56-67.

8.14.2 Explaining Tree Models with Path-Dependent Feature Perturbation Tree SHAP

Note

To enable SHAP support, you may need to run

```
pip install alibi[shap]
```

```
[ ]: # shap.summary_plot currently doesn't work with matplotlib>=3.6.0,
# see bug report: https://github.com/slundberg/shap/issues/2687
!pip install matplotlib==3.5.3
```


Introduction

This example shows how to apply path-dependent feature perturbation Tree SHAP to compute shap values exactly for an `xgboost` model fitted to the `Adult` dataset (binary classification task). An example of how to decompose the contribution of any given feature into a main effect and interactions with other features is also presented.

This example will use the `xgboost` library, which can be installed with:

```
[ ]: !pip install xgboost

[1]: import pickle
import shap
shap.initjs()

import numpy as np
import matplotlib.pyplot as plt
import xgboost as xgb

from alibi.datasets import fetch_adult
from alibi.explainers import TreeShap
from functools import partial
from itertools import product, zip_longest
from scipy.special import expit
invlogit=expit
from sklearn.metrics import accuracy_score, confusion_matrix

from timeit import default_timer as timer

<IPython.core.display.HTML object>
```

Data preparation

Load and split

The `fetch_adult` function returns a `Bunch` object containing the features, targets, feature names and a mapping of categorical variables to numbers.

```
[2]: adult = fetch_adult()
adult.keys()

[2]: dict_keys(['data', 'target', 'feature_names', 'target_names', 'category_map'])

[3]: data = adult.data
target = adult.target
target_names = adult.target_names
feature_names = adult.feature_names
category_map = adult.category_map
```

Note that for your own datasets you can use the utility function `gen_category_map` imported from `alibi.utils` to create the category map.

```
[4]: np.random.seed(0)
data_perm = np.random.permutation(np.c_[data, target])
```

(continues on next page)

(continued from previous page)

```
data = data_perm[:, :-1]
target = data_perm[:, -1]
```

```
[5]: idx = 30000
X_train, y_train = data[:idx, :], target[:idx]
X_test, y_test = data[idx+1:, :], target[idx+1:]
```

xgboost wraps arrays using DMatrix objects, optimised for both memory efficiency and training speed.

```
[6]: def wrap(arr):
    return np.ascontiguousarray(arr)

dtrain = xgb.DMatrix(
    wrap(X_train),
    label=wrap(y_train),
    feature_names=feature_names,
)

dtest = xgb.DMatrix(wrap(X_test), label=wrap(y_test), feature_names=feature_names)
```

Finally, a matrix that contains the raw string values for categorical variables (used for display) is created:

```
[7]: def _decode_data(X, feature_names, category_map):
    """
    Given an encoded data matrix `X` returns a matrix where the
    categorical levels have been replaced by human readable categories.
    """

    X_new = np.zeros(X.shape, dtype=object)
    for idx, name in enumerate(feature_names):
        categories = category_map.get(idx, None)
        if categories:
            for j, category in enumerate(categories):
                encoded_vals = X[:, idx] == j
                X_new[encoded_vals, idx] = category
        else:
            X_new[:, idx] = X[:, idx]

    return X_new

decode_data = partial(_decode_data, feature_names=feature_names, category_map=category_
↪map)
```

```
[8]: X_display = decode_data(X_test)
```

```
[9]: X_display
```

```
[9]: array([[52, 'Private', 'Associates', ..., 0, 60, 'United-States'],
        [21, 'Private', 'High School grad', ..., 0, 20, 'United-States'],
        [43, 'Private', 'Dropout', ..., 0, 50, 'United-States'],
        ...,
        [23, 'Private', 'High School grad', ..., 0, 40, 'United-States'],
```

(continues on next page)

(continued from previous page)

```
[45, 'Local-gov', 'Doctorate', ..., 0, 45, 'United-States'],
[25, 'Private', 'High School grad', ..., 0, 48, 'United-States']],
dtype=object)
```

Model definition

The model fitted in the xgboost fitting example will be explained. The confusion matrix of this model is shown below:

```
[10]: def plot_conf_matrix(y_test, y_pred, class_names):
    """
    Plots confusion matrix. Taken from:
    http://queirozf.com/entries/visualizing-machine-learning-models-examples-with-scikit-learn-and-matplotlib
    """

    matrix = confusion_matrix(y_test, y_pred)

    # place labels at the top
    plt.gca().xaxis.tick_top()
    plt.gca().xaxis.set_label_position('top')

    # plot the matrix per se
    plt.imshow(matrix, interpolation='nearest', cmap=plt.cm.Blues)

    # plot colorbar to the right
    plt.colorbar()

    fmt = 'd'

    # write the number of predictions in each bucket
    thresh = matrix.max() / 2.
    for i, j in product(range(matrix.shape[0]), range(matrix.shape[1])):
        # if background is dark, use a white number, and vice-versa
        plt.text(j, i, format(matrix[i, j], fmt),
                horizontalalignment="center",
                color="white" if matrix[i, j] > thresh else "black")

    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names, rotation=45)
    plt.yticks(tick_marks, class_names)
    plt.tight_layout()
    plt.ylabel('True label', size=14)
    plt.xlabel('Predicted label', size=14)
    plt.show()

def predict(xgb_model, dataset, proba=False, threshold=0.5):
    """
    Predicts labels given a xgboost model that outputs raw logits.
    """
```

(continues on next page)

(continued from previous page)

```

y_pred = model.predict(dataset) # raw logits are predicted
y_pred_proba = invlogit(y_pred)
if proba:
    return y_pred_proba
y_pred_class = np.zeros_like(y_pred)
y_pred_class[y_pred_proba >= threshold] = 1 # assign a label

return y_pred_class

```

```

[11]: model = xgb.Booster()
      model.load_model('assets/adult_xgb.mdl')

```

```

[12]: y_pred_train = predict(model, dtrain)
      y_pred_test = predict(model, dtest)

```

```

[13]: plot_conf_matrix(y_test, y_pred_test, target_names)

```

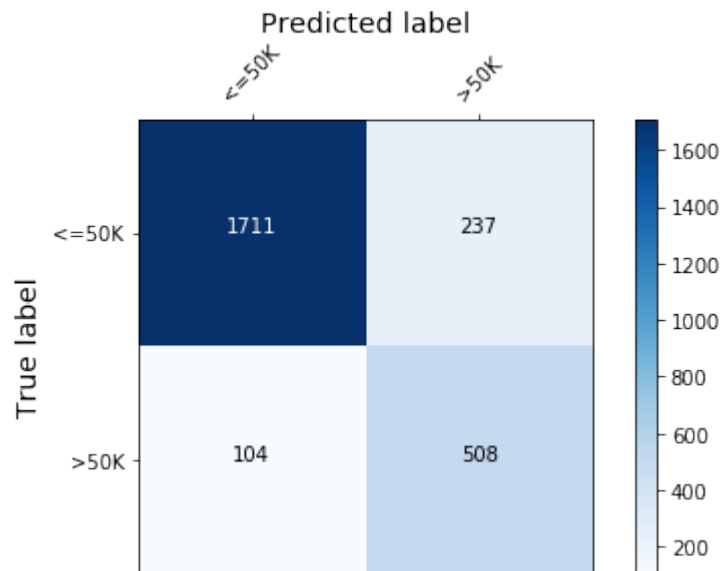


Figure 1: Model confusion matrix

```

[14]: print(f'Train accuracy: {round(100*accuracy_score(y_train, y_pred_train), 4)} %.')
      print(f'Test accuracy: {round(100*accuracy_score(y_test, y_pred_test), 4)}%.')

```

```

Train accuracy: 87.75 %.
Test accuracy: 86.6797%.

```

Explaining xgboost via global feature importance

Locally, one could interpret an outcome predicted by a decision tree by analysing the path followed by the sample through the tree (known as the *decision path*). However, for xgboost the final decision depends on the number of boosting rounds so this technique is not practical. Moreover, this approach only informs one about *which features* factored in the decision of the algorithm but nothing about the relative importance of the features. Such a view can only be obtained at a global level, for example, by combining information from decision paths of all ensemble members. The xgboost library offers the following measures of feature importance for a feature:

- `weight` - the number of times a feature is used to split the data across all trees
- `gain` - the average gain (that is, contribution to the model output) across all splits the feature is used in
- `cover(I)` - the average coverage across all splits the feature is used in
- `total_gain` - the total gain across all splits the feature is used in
- `total_cover` - the total coverage across all splits the feature is used in.

Therefore, one is first faced with the task of choosing *a notion of feature importance* before interpreting their model. As shown below, different notions of feature importance lead to different explanations for the same model.

```
[15]: def _get_importance(model, measure='weight'):
    """
    Retrieves the feature importances from an xgboost
    models, measured according to the criterion `measure`.
    """

   imps = model.get_score(importance_type=measure)
    names, vals = list(imps.keys()), list(imps.values())
    sorter = np.argsort(vals)
    s_names, s_vals = tuple(zip(*[(names[i], vals[i]) for i in sorter]))

    return s_vals[::-1], s_names[::-1]

def plot_importance(feats_imp, feat_names, ax=None, **kwargs):
    """
    Create a horizontal barchart of feature effects, sorted by their magnitude.
    """

    left_x, step, right_x = kwargs.get("left_x", 0), kwargs.get("step", 50), kwargs.get(
        "right_x")
    xticks = np.arange(left_x, right_x, step)
    xlabel = kwargs.get("xlabel", 'Feature effects')
    xposfactor = kwargs.get("xposfactor", 1)
    textfont = kwargs.get("text_fontsize", 25) # 16
    yticks_fontsize = kwargs.get("yticks_fontsize", 25)
    xlabel_fontsize = kwargs.get("xlabel_fontsize", 30)
    textxpos = kwargs.get("textxpos", 60)
    textcolor = kwargs.get("textcolor", 'white')

    if ax:
        fig = None
    else:
        fig, ax = plt.subplots(figsize=(10, 5))
```

(continues on next page)

(continued from previous page)

```

y_pos = np.arange(len(feats_imp))
ax.barh(y_pos, feat_imp)
ax.set_yticks(y_pos)
ax.set_yticklabels(feats_names, fontsize=yticks_fontsize)
ax.set_xticklabels(xticks, fontsize=30, rotation=45)
ax.invert_yaxis() # labels read top-to-bottom
ax.set_xlabel(xlabel, fontsize=xlabel_fontsize)
ax.set_xlim(left=left_x, right=right_x)

for i, v in enumerate(feats_imp):
    # if v<0:
        textxpos = xposfactor*textxpos
        ax.text(v - textxpos, i + .25, str(round(v, 3)), fontsize=textfont,
        color=textcolor)
    return ax, fig

get_importance = partial(_get_importance, model)

```

To demonstrate this, the feature importances obtained when the measures of importance are set to `weight`, `total_gain` and `gain` are plotted below. The difference between the latter two is that the decrease in loss due to a feature is reported as a sum (`total_gain`) and as an average across the splits (`gain`).

```

[16]: imp_by_weight_v, imp_by_weight_n = get_importance()
      imp_by_gain_v, imp_by_gain_n = get_importance(measure='total_gain')
      imp_by_a_gain_v, imp_by_a_gain_n = get_importance(measure='gain')

```

```

[17]: fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(62, 13))
      plot_importance(imp_by_weight_v, imp_by_weight_n, ax=ax1, xlabel='Feature effects_
      (weights)', textxpos=45, right_x=1000, step=200 )
      plot_importance(imp_by_gain_v, imp_by_gain_n, ax=ax2, xlabel='Feature effects (total_
      gain)', textxpos=5, right_x=65000, step=10000, textcolor='black')
      plot_importance(imp_by_a_gain_v, imp_by_a_gain_n, ax=ax3, xlabel='Feature effects (gain)
      ', textxpos=0, right_x=250, step=50, textcolor='black')

```

```

[17]: (<matplotlib.axes._subplots.AxesSubplot at 0x7feb8944e650>, None)

```

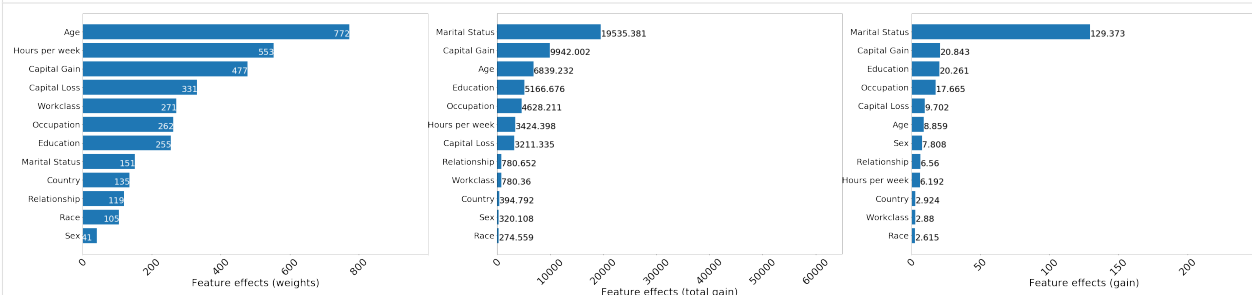


Figure 2: Feature importances as measured by the total number of splits (left), total loss decrease due to feature (middle) and average decrease in loss due to splitting on a particular feature (right)

When using the `weight` criterion for feature importance, all the continuous variables are ranked above categorical ones in terms of feature importance. This occurs because these continuous variables can be split multiple times at different levels in the tree, whereas binary variables such as `Sex` can only be used to partition the data once, so the expected number of splits is smaller for such a variable. To avoid such biases, the feature importance can be quantified by using the total and average gain in information (or, equivalently, decrease in objective). Although the `Marital Status`

feature was used to partition the data only 151 times, it contributed the most to decreasing the loss, both across the entire ensemble and when averaged across the splits.

In general, the notion of importance should balance the information gain from making a split on a particular feature with how frequently this feature is used for splitting. Features such as `Age` may have a large cumulative gain courtesy to them being split on multiple times, but on average they may contribute less to the outcome compared to other features such as `Capital Gain` which are also split on significant number of times.

However, despite mitigating some of the shortcomings of the split-frequency feature importance, the gain notion of feature-importance suffers from *lack of consistency*, a property that allows one to compare feature effects across models. The interested reader is referred to [this](#) example (page 22) published by Lundberg et al. for details. Such a problem can be mitigated by defining the notion of feature importance with respect to Shapley values, which are consistent as well as faithful to the model (locally).

Explaining xgboost with path-dependent Tree SHAP: global knowledge from local explanations

As described in the [overview](#), the path-dependent feature perturbation Tree SHAP algorithm uses node-level statistics (cover) extracted from the training data in order to estimate the effect of missing features on the model output. Since tree structures also support efficient computation of the model outputs for all possible subsets of missing features, the use of tree paths makes exact shap value estimation possible *without* a background dataset. In contrast, algorithms such as Kernel SHAP use a background dataset to *approximate* shap values while interventional feature perturbation Tree SHAP uses a background dataset to compute the effect of missing features on function output and *exactly* computes the feature contributions given these values.

```
[18]: path_dependent_explainer = TreeShap(model, model_output='raw', task='classification')
      path_dependent_explainer.fit() # does not require background_data
```

Setting feature_perturbation = "tree_path_dependent" because no background data was given.

Predictor returned a scalar value. Ensure the output represents a probability or decision score as opposed to a classification label!

```
[18]: TreeShap(meta={
      'name': 'TreeShap',
      'type': ['whitebox'],
      'task': 'classification',
      'explanations': ['local', 'global'],
      'params': {'summarise_background': False, 'kwargs': {}}
    })
```

Note that the `model_output` kwarg was set to `raw`, to indicate the fact that the model outputs log-odds ratios(2). This is the only option supported at this moment by this algorithm.

```
[19]: path_dependent_explanation = path_dependent_explainer.explain(X_test)
      path_dependent_shap_values = path_dependent_explanation.shap_values[0]
```

The shap values computed in this way have the local accuracy property, as expected. That is, they sum to the difference between the model output to be explained and the reference value.

```
[20]: np.max(np.abs(model.predict(dtest) - path_dependent_explainer.expected_value - path_
      ↪dependent_shap_values.sum(1)))
```

```
[20]: 0.5000074921901536
```

The features which are most important for the predicting whether an individual makes an income greater than \\$50,

000 are shown in Figure 3, where the feature importance of feature j is defined as:

$$I_j = \frac{1}{N} \sum_{i=1}^N |\phi_{i,j}|.$$

Here N is the size of the explained dataset. According to this criterion, the `Marital Status` feature seems to be the most important, followed by features such as `Age` or `Capital Gain`. This global view does not provide information about the *direction* of the effect at individual level (i.e., whether the prediction that an individual earns more than \$50,000 is affected positively or negatively by a particular feature), the *magnitude* of the effect at individual level (i.e., whether the `Marital Status` feature, the most important globally, has a significant impact on the prediction about each individual) or the *prevalence* of a particular effect (how many members of the population are affected in similar ways by a particular feature).

```
[21]: shap.summary_plot(path_dependent_shap_values, X_test, feature_names, plot_type='bar')
```

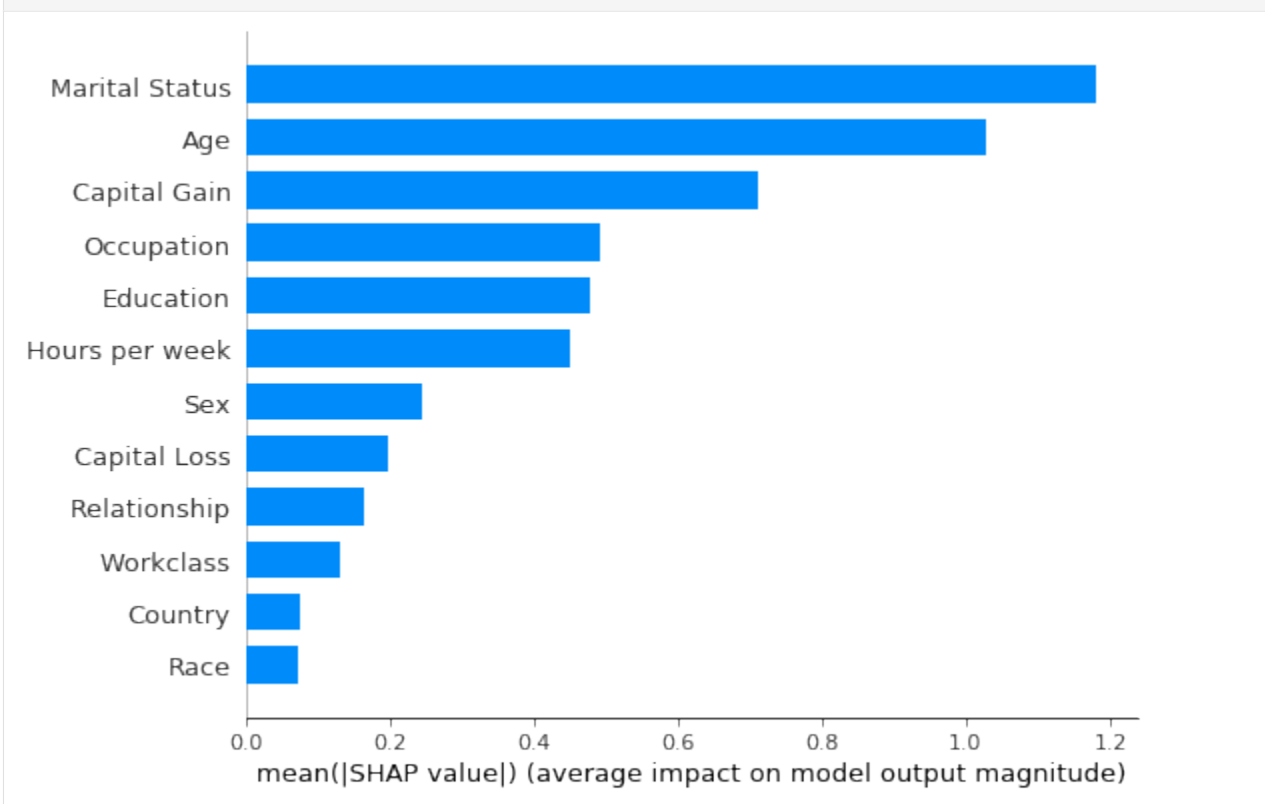


Figure 3: Most important features as predicted by the path-dependent perturbation Tree SHAP algorithm

To answer such questions, the same feature ranking can be displayed in a *summary plot* (Figure 4), which is an aggregation of local explanations. Note that at each feature, points with the same shap value pile up to show density.

```
[22]: shap.summary_plot(path_dependent_shap_values, X_test, feature_names, class_names=target_
↪ names)
```

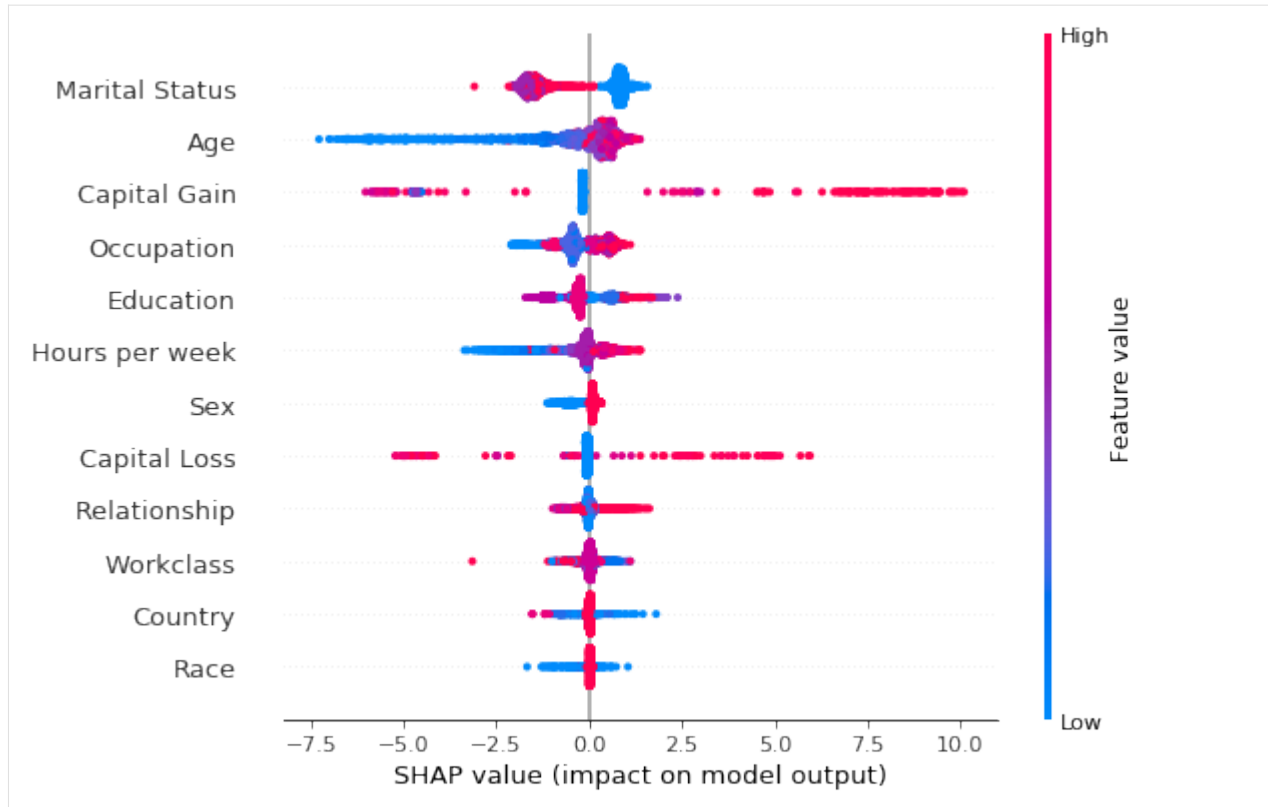



Figure 4: Summary plot of the path-dependent perturbation Tree SHAP explanations for the test set

```
[23]: from collections import Counter
```

```
feat_name = 'Marital Status'
decode_dict = {}
for i, val in enumerate(category_map[feature_names.index(feat_name)]):
    decode_dict[i] = val
print("Mapping of feature encoded values to readable values", decode_dict)
```

```
Mapping of feature encoded values to readable values {0: 'Married', 1: 'Never-Married', 2: 'Separated', 3: 'Widowed'}
```

The summary plot shows that being married increases the odds of making more than \$50,000 and that, with few exceptions, being widowed decreases the odds of making an income above this threshold. Despite having a significant effect in aggregate, the Age feature does not affect all individuals as significantly: the impact on the prediction of this feature can be significantly negative for young individuals, making it unlikely that young individuals will be predicted to earn more than \$50,000. However, while in general elderly tend to be more wealthy, the impact of this feature can be less significant compared to other “less important” features such as Capital Loss, Capital Gain or Education.

The tails in the summary plot of the Capital Loss feature indicate that while this feature is less important than Education or Sex as far as the global model behaviour is concerned, for specific individuals this feature can be a stronger predictor of the income class than the aforementioned features. This granularity in explanations is beyond the reach of traditional methods for tree interpretability.

The vertical spread in the summary plots is indicative of feature interactions, which can be identified approximately, as described in this example, through the shap dependence plot. The Model explanations with Shapley interaction values section shows that Tree SHAP supports exact computation of *Shapley interaction values* which allow attributing a change in an outcome not only to the features, but also to first order interactions between features.

```
[24]: def _dependence_plot(features, shap_values, dataset, feature_names, category_map,
↳ display_features=None, **kwargs):
    """
    Plots dependence plots of specified features in a grid.

    features: List[str], List[Tuple[str, str]]
        Names of features to be plotted. If List[str], then shap
        values are plotted as a function of feature value, coloured
        by the value of the feature determined to have the strongest
        interaction (empirically). If List[Tuple[str, str]], shap
        interaction values are plotted.
    display_features: np.ndarray, N x F
        Same as dataset, but contains human readable values
        for categorical levels as opposed to numerical values
    """

    def _set_fonts(fig, ax, fonts=None, set_cbar=False):
        """
        Sets fonts for axis labels and colobar.
        """

        ax.xaxis.label.set_size(xlabelfontsize)
        ax.yaxis.label.set_size(ylabelfontsize)
        ax.tick_params(axis='x', labelsizex=xlabelfontsize)
        ax.tick_params(axis='y', labelsizex=ylabelfontsize)
        if set_cbar:
            fig.axes[-1].tick_params(labelsizex=cbarlabelfontsize)
            fig.axes[-1].tick_params(labelrotation=cbartickrotation)
            fig.axes[-1].yaxis.label.set_size(cbarlabelfontsize)

    # parse plotting args
    figsize = kwargs.get("figsize", (15, 10))
    nrows = kwargs.get('nrows', len(features))
    ncols = kwargs.get('ncols', 1)
    xlabelfontsize = kwargs.get('xlabelfontsize', 14)
    xtickfontsize = kwargs.get('xtickfontsize', 11)
    ylabelfontsize = kwargs.get('ylabelfontsize', 14)
    ytickfontsize = kwargs.get('ytickfontsize', 11)
    cbartickfontsize = kwargs.get('cbartickfontsize', 14)
    cbartickrotation = kwargs.get('cbartickrotation', 10)
    cbarlabelfontsize = kwargs.get('cbarlabelfontsize', 14)
    rotation_orig = kwargs.get('xticklabelrotation', 25)

    alpha = kwargs.get("alpha", 1)
    x_jitter_orig = kwargs.get("x_jitter", 0.8)
    grouped_features = list(zip_longest(*[iter(features)] * ncols))

    fig, axes = plt.subplots(nrows, ncols, figsize=figsize)
    if nrows == len(features):
        axes = list(zip_longest(*[iter(axes)] * 1))
```

(continues on next page)

(continued from previous page)

```

for i, (row, group) in enumerate(zip(axes, grouped_features), start=1):
    # plot each feature or interaction in a subplot
    for ax, feature in zip(row, group):
        # set x-axis ticks and labels and x-jitter for categorical variables
        if not feature:
            continue
        if isinstance(feature, list) or isinstance(feature, tuple):
            feature_index = feature_names.index(feature[0])
        else:
            feature_index = feature_names.index(feature)
        if feature_index in category_map:
            ax.set_xticks(np.arange(len(category_map[feature_index])))
            if i == nrows:
                rotation = 90
            else:
                rotation = rotation_orig
            ax.set_xticklabels(category_map[feature_index], rotation=rotation,
↪ fontsize=22)
            x_jitter = x_jitter_orig
        else:
            x_jitter = 0

        shap.dependence_plot(feature,
                             shap_values,
                             dataset,
                             feature_names=feature_names,
                             display_features=display_features,
                             interaction_index='auto',
                             ax=ax,
                             show=False,
                             x_jitter=x_jitter,
                             alpha=alpha
                             )

    if i != nrows:
        ax.tick_params('x', labelrotation=rotation_orig)
    _set_fonts(fig, ax, set_cbar=True)

plot_dependence = partial(
    _dependence_plot,
    feature_names=feature_names,
    category_map=category_map,
)

```

The dependence plots (Figure 5, below) reveal that the strongest interaction of the Marital Status shap values are due to the Hours per week variable. Although the odds for earning in excess of \$50, 000 are against people who are not married or have separated, they tend to be more favourable for individuals working long hours.

As far as Age is concerned, the odds of earning more increase as a person ages, and, in general, this variable is used by the model to assign individuals to a lower income class. People in their 30s-60s are thought to be more likely to make an income over \$50, 000 if their capital gains are high. Interestingly, for people over 60, high capital gains have a large negative contribution to the odds of making large incomes, a pattern that is perhaps not intuitive.

As far as the Hours per week is concerned, one sees that older people working no to few hours a week are predicted better odds for making a larger income, and that, up to a certain threshold (of approximately 60 hours), working more

than 20 hours increases the odds of a > \$50, 000 prediction for all ages.

Finally, note that not knowing the occupation hurts the odds of predicting a high income. No significant interactions between the sex of the individual (males in red), their occupation and their predicted odds are observed with the exception of, perhaps, Admin and Blue Collar groups.

Warning

For the following plots to run the matplotlib version needs to be <3.5.0. This is because of an upstream issue of how the `shap.dependence_plot` function is handled in the `shap` library. An issue tracking it can be found [here](#).

```
[25]: plot_dependence(
      ['Marital Status', 'Age', 'Hours per week', 'Occupation'],
      path_dependent_shap_values,
      X_test,
      alpha=0.5,
      x_jitter=0.8,
      nrows=2,
      ncols=2,
    )
```

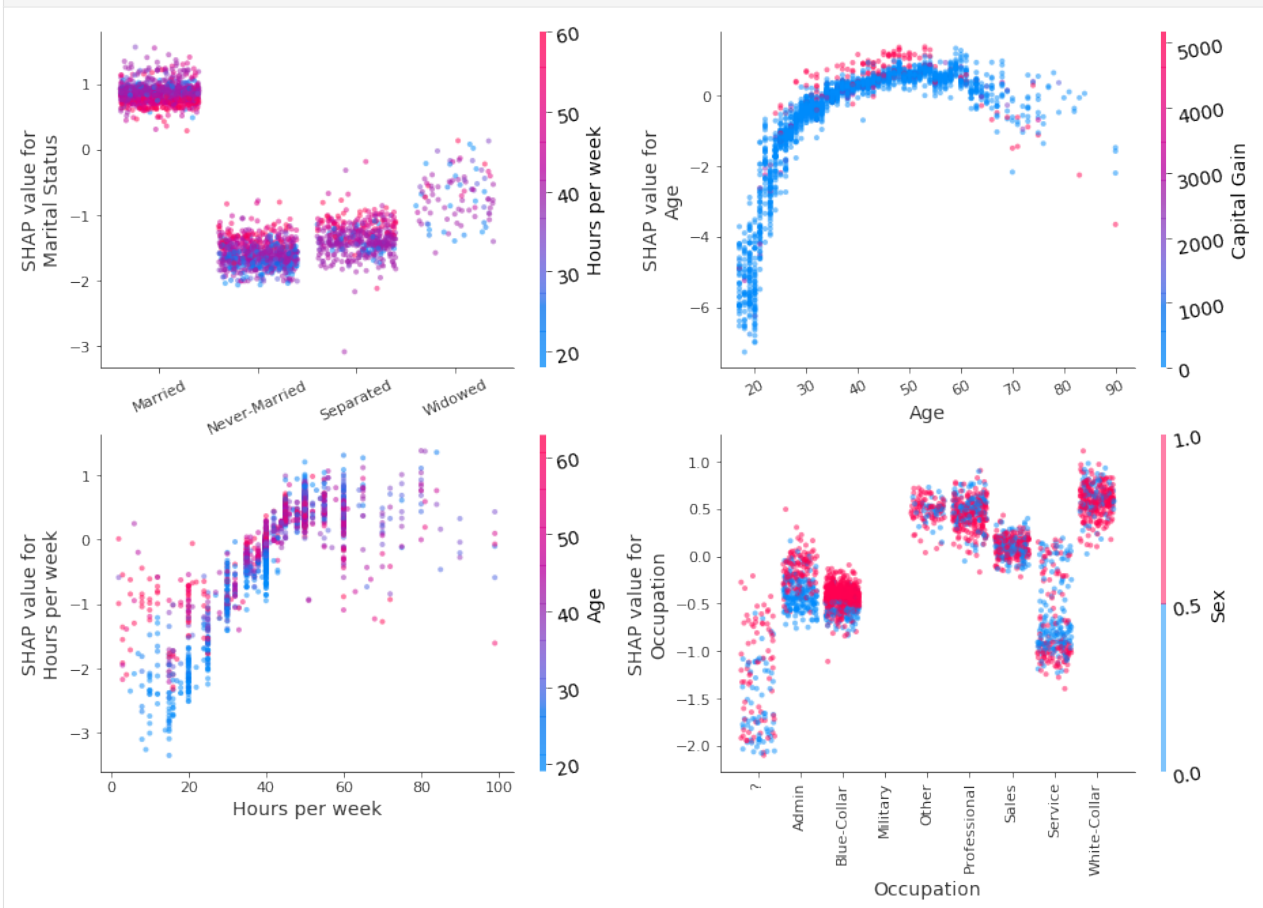


Figure 5: Decision plot of Marital Status, Age, Hours per week, Occupation features using the path-dependent perturbation Tree SHAP algorithm. Colouring is according to the value of the variable estimated to have the strongest interaction with the plotted variable. Jitter in the x direction has been applied to categorical variables to improve readability.

Performing local explanations across multiple instances efficiently can provide insight into how features contributed to misclassifications and the most common causes of misclassification. This can be achieved by performing a similar analysis for those individuals whose income was predicted below \$50, 000 but who are known to make an income in excess of this threshold.

```
[26]: # identify false negatives
misclassified = (np.logical_and(y_test == 1, y_pred_test == 0)).nonzero()[0]
X_misclassified = X_test[misclassified]
# explain the predictions
shap_vals_misclassified = path_dependent_shap_values[misclassified, :]
```

The summary plot indicates that the feature with the most impact on misclassification is Marital Status and that the model does not correctly capture the fact that individuals who were never married, widowed or separated can also make high incomes.

```
[27]: shap.summary_plot(shap_vals_misclassified, X_misclassified, feature_names )
```

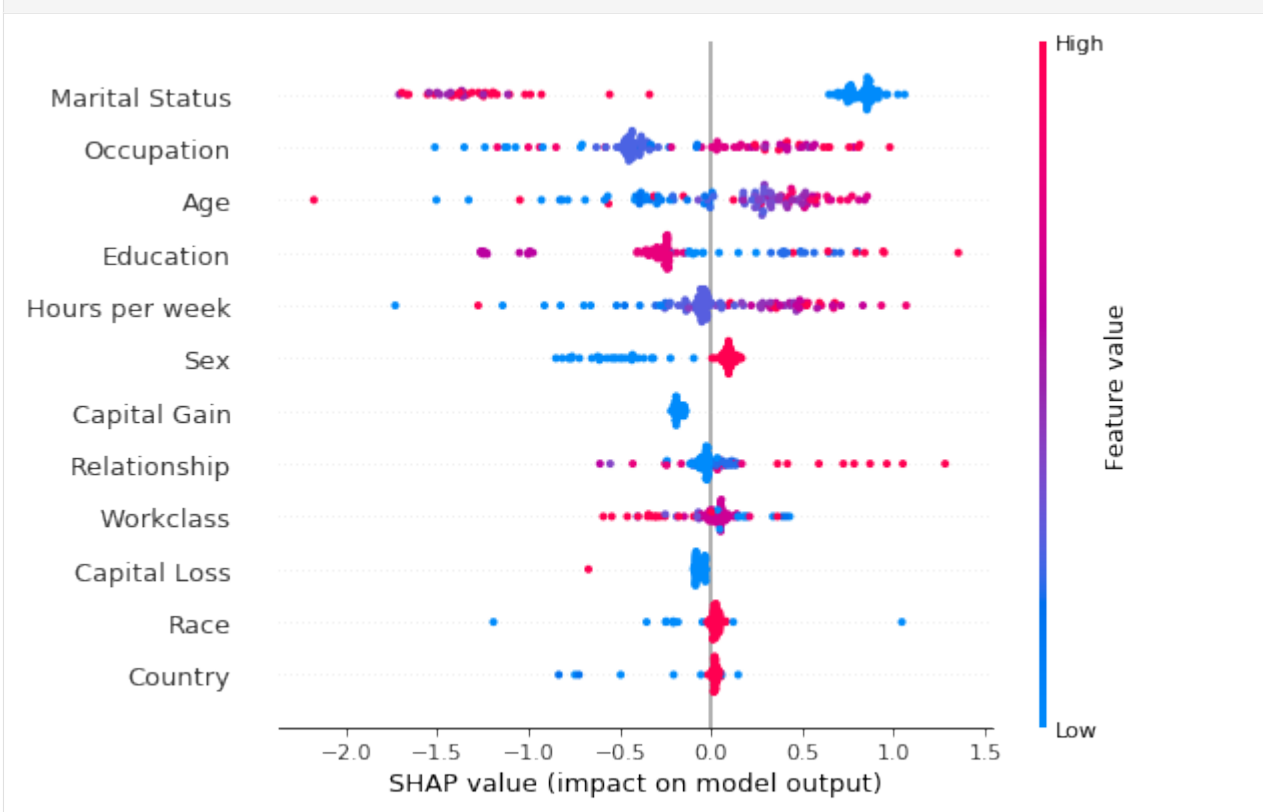


Figure 6: Summary plot of path-dependent perturbation Tree SHAP explanations for individuals misclassified as earning less than \$50, 000.

```
[28]: X_misclassified_display = decode_data(X_misclassified)
plot_dependence(
    ['Marital Status', 'Age', 'Sex', 'Race', 'Occupation', 'Education'],
    shap_vals_misclassified,
    X_misclassified,
    display_features=X_misclassified_display,
    rotation=33,
    figsize=(47.5, 22),
    alpha=1,
```

(continues on next page)

(continued from previous page)

```

x_jitter=0.5,
nrows=3,
ncols=2,
xlabelfontsize=24,
xtickfontsize=20,
xticklabelrotation=0,
ylabelfontsize=24,
ytickfontsize=21,
cbarlabelfontsize=22,
cbartickfontsize=20,
cbartickrotation=0,

```

)

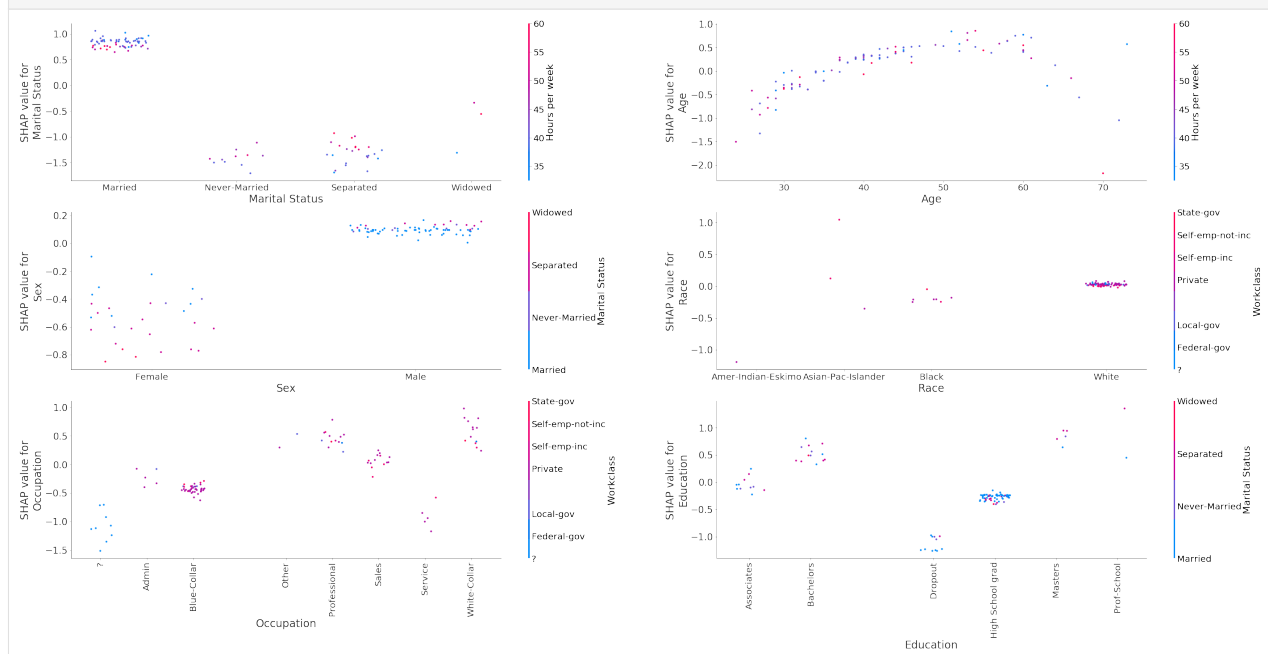


Figure 7: Decision plots of the variables Marital Status, Age, Sex, Race, Occupation, Education using the path-dependent Tree SHAP algorithm for individuals misclassified as earning less than \$50,000.

Analysing the plots above reveals that some of the patterns that can lead to misclassification are:

- individuals are not married or are divorced/widowed
- individuals below 40 years old are expected to earn less, across all occupation categories
- individuals are female; being single further increases the odds against the high income class
- racial bias does not seem to be one of the drivers of misclassification, although we can see that for Black people the contribution is slightly negative whereas for white people the contribution is zero
- individuals being Blue-Collar workers, working in Admin jobs, the Service industry or individuals whose occupation is unknown
- individuals having dropped out of education or being high school graduates

Model explanations with Shapley interaction values

As described in the algorithm [overview](#), path-dependent feature perturbation Tree Shap can attribute a change in outcome not only to the M input features, but to the M features and the first-order interactions between them. For each instance to be explained, a tensor of $M \times M$ numbers is returned. The diagonal of this tensor, indexed by (i, i) , represents the *main effects* (i.e., due to the feature itself) whereas the off-diagonal terms indexed by (i, j) represent the *interaction between the i th and the j th feature in the input*. Summing along the rows of an entry in the Shapley interaction values tensor yields the M shap values for that instance. Note that the interaction value is split equally between each feature so the returned matrix is symmetric; the total interaction effect between feature i and j is therefore obtained by adding the two symmetric entries (i, j) and (j, i) .

```
[29]: shap_interactions_explanation = path_dependent_explainer.explain(X_test,
↳ interactions=True)
```

```
[30]: shap_interactions_values = shap_interactions_explanation.shap_interaction_values[0]
```

Plots of the interactions between the features Age, Sex, Education and Occupation with Capital Gain are shown below.

```
[31]: plot_dependence(
    [('Age', 'Capital Gain'),
     ('Sex', 'Capital Gain'),
     ('Education', 'Capital Gain'),
     ('Occupation', 'Capital Gain'),
    ],
    shap_interactions_values,
    X_test,
    figsize=(30,16.5),
    rotation=15,
    ncols=2,
    nrows=2,
    display_features=X_display,
    xtickfontsize=20,
    xlabelfontsize=20,
    ylabelfontsize=20,
    ytickfontsize=17,
    cbarlabelfontsize=20,
    cbartickfontsize=18,
)
```

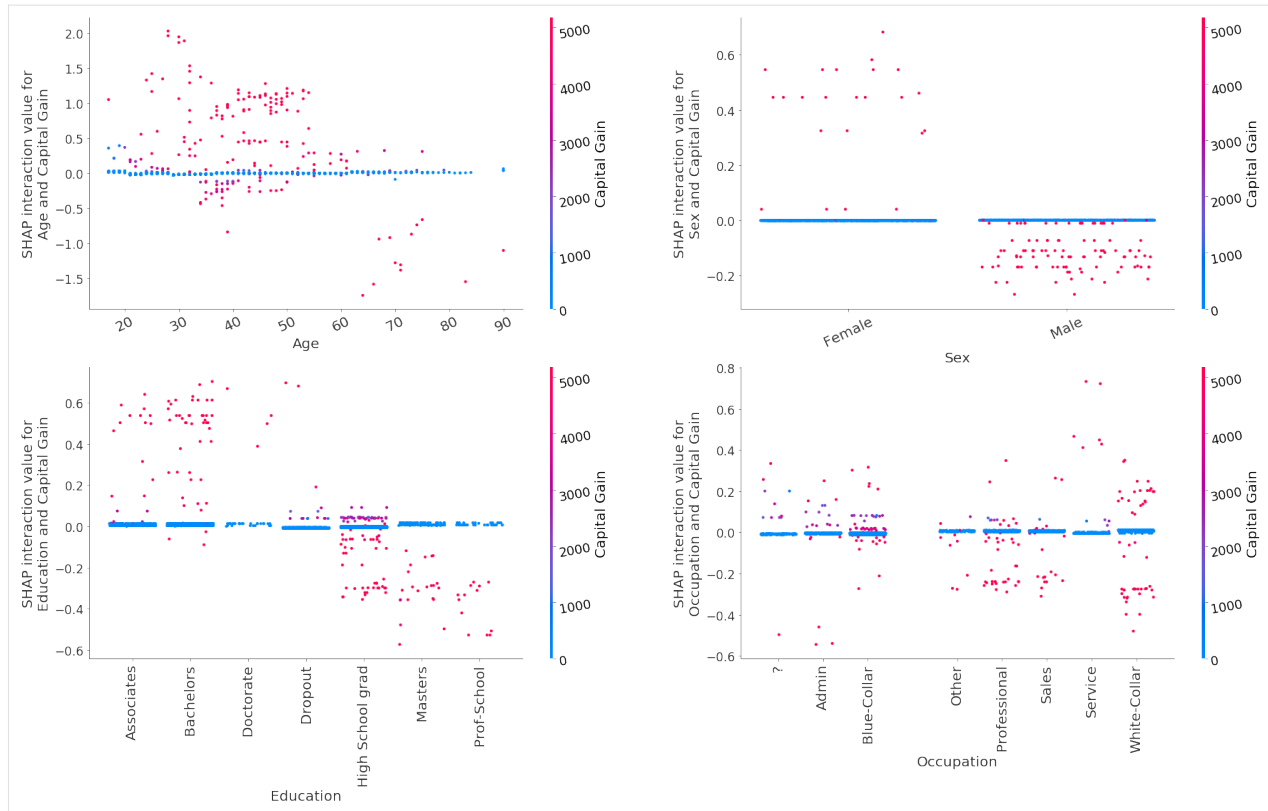


Figure 11: Shap interaction values for the features Age, Sex, Education and Occupation with Capital Gain

The model has captured the following patterns:

- The interaction between Age and Capital gain increases the odds of predicting an income >\$50,000 for most individuals below 60 years old but significantly decreases the odds for individuals above 60 years old. This interaction has no effect when the individuals don't have any capital gains
- For females, capital gains generally increase the prediction odds while for males they decrease them, although these latter interactions are much smaller in magnitude
- Having a capital gain and education level at Masters and Prof-School or High School grad decreases the prediction odds for higher income
- For most individuals in occupation categories Professional and Sales, high capital gains slightly reduce the odds of predicting >\$50,000. For White-Collar individuals, high capital gain can both increase or decrease the odds.

The `plot_decomposition` function can be used to decompose the shap values of a particular feature into a set of shap values that do not account for the interaction with a specific feature and the interaction values with that specific feature, as shown below. This is depicted in Figure 12.

```
[32]: def plot_decomposition(feature_pair, shap_interaction_vals, features, feat_names,
    ↳ display_features=None, **kwargs):
    """
    Given a list containing two feature names (`feature_pair`), an  $n_{\text{instances}} \times n_{\text{features}}$ 
    ↳ features  $\times n_{\text{features}}$  tensor
    of shap interaction values (`shap_interaction_vals`), an  $n_{\text{instances}} \times n_{\text{features}}$ 
    ↳ (`features`) tensor of
    feature values and a list of feature names (which assigns a name to each column of
```

(continues on next page)

(continued from previous page)

```

↪ `features`), this function
    plots:
        - left: shap values for feature_pair[0] coloured by the value of feature_pair[1]
        - middle: shap values for feature_pair[0] after subtracting the interaction with_
↪ feature_pair[1]
        - right: the interaction values between feature_pair[0] and feature_pair[1],_
↪ which are subtracted
        from the left plot to get the middle plot

    NB: `display_features` is the same shape as `features` but should contain the raw_
↪ categories for categorical
        variables so that the colorbar can be discretised and the category names displayed_
↪ alongside the colorbar.
    """

    def _set_fonts(fig, ax, fonts=None, set_cbar=False):
        """
        Sets fonts for axis labels and colorbar.
        """

        ax.xaxis.label.set_size(xlabelfontsize)
        ax.yaxis.label.set_size(ylabelfontsize)
        ax.tick_params(axis='x', labelsizex=xlabelfontsize)
        ax.tick_params(axis='y', labelsizex=ylabelfontsize)
        if set_cbar:
            fig.axes[-1].tick_params(labelsize=cbartickfontsize)
            fig.axes[-1].yaxis.label.set_size(cbarlabelfontsize)

    # parse plotting args
    xlabelfontsize = kwargs.get('xlabelfontsize', 21)
    ylabelfontsize = kwargs.get('ylabelfontsize', 21)
    cbartickfontsize = kwargs.get('cbartickfontsize', 16)
    cbarlabelfontsize = kwargs.get('cbarlabelfontsize', 21)
    xtickfontsize = kwargs.get('xtickfontsize', 20)
    ytickfontsize = kwargs.get('ytickfontsize', 20)
    alpha = kwargs.get('alpha', 0.7)
    figsize = kwargs.get('figsize', (44, 10))
    ncols = kwargs.get('ncols', 3)
    nrows = kwargs.get('nrows', 1)
    # compute shap values and shap values without interaction
    feat1_idx = feat_names.index(feature_pair[0])
    feat2_idx = feat_names.index(feature_pair[1])
    # shap values
    shap_vals = shap_interaction_vals.sum(axis=2)
    # shap values for feat1, all samples
    shap_val_ind1 = shap_interaction_vals[:, feat1_idx].sum(axis=1)
    # shap values for (feat1, feat2) interaction
    shap_int_ind1_ind2 = shap_interaction_vals[:, feat2_idx, feat1_idx]
    # subtract effect of feat2
    shap_val_minus_ind2 = shap_val_ind1 - shap_int_ind1_ind2
    shap_val_minus_ind2 = shap_val_minus_ind2[:, None]

```

(continues on next page)

(continued from previous page)

```

# create plot

fig, (ax1, ax2, ax3) = plt.subplots(nrows, ncols, figsize=figsize)

# plot the shap values including the interaction
shap.dependence_plot(feature_pair[0],
                      shap_vals,
                      features,
                      display_features = display_features,
                      feature_names=feat_names,
                      interaction_index=feature_pair[1],
                      alpha=alpha,
                      ax=ax1,
                      show=False)
_set_fonts(fig, ax1, set_cbar=True)

# plot the shap values excluding the interaction
shap.dependence_plot(0,
                      shap_val_minus_ind2,
                      features[:, feat1_idx][:, None],
                      feature_names=[feature_pair[0]],
                      interaction_index=None,
                      alpha=alpha,
                      ax=ax2,
                      show=False,
                      )
ax2.set_ylabel(f' Shap value for {feature_pair[0]} \n wo {feature_pair[1]} ↵
↵interaction')
_set_fonts(fig, ax2)

# plot the interaction value
shap.dependence_plot(feature_pair,
                      shap_interaction_vals,
                      features,
                      feature_names=feat_names,
                      display_features=display_features,
                      interaction_index='auto',
                      alpha=alpha,
                      ax=ax3,
                      show=False,
                      )
_set_fonts(fig, ax3, set_cbar=True)

```

```

[33]: feature_pair = ('Age', 'Capital Gain')
      plot_decomposition(
          feature_pair,
          shap_interactions_values,
          X_test,
          feature_names,
          display_features=X_display,
      )

```

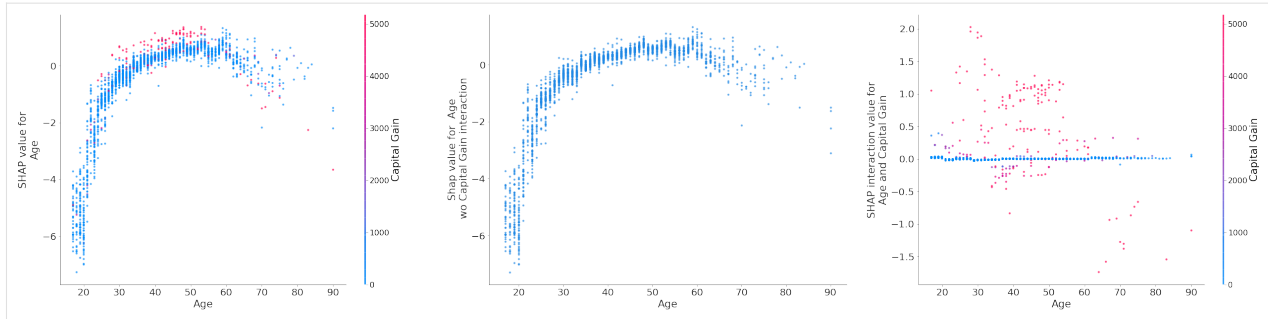


Figure 12: A decomposition of the shap values for Age (left) into shap values for Age excluding the Capital Gain interaction (middle). The total interaction between Age and Capital Gain shown on right.

Model explanations using xgboost predict method

The `xgboost` library implements an optimised version of the path-dependent feature perturbation algorithm, which is also internally used by the `shap` library. `xgboost` also provides an optimised algorithm for computing the shap interaction values.

The `predict` method can output the shap values if called as follows:

```
[34]: xgb_shap_vals = model.predict(dtest, pred_contribs=True)
```

```
[35]: print(f"shap values shape: {xgb_shap_vals.shape}")
```

```
shap values shape: (2560, 13)
```

Note that there are only 12 features in the dataset. The last column is the expected value with respect to which the feature contributions are computed.

One can also estimate the shap interaction values as follows:

```
[36]: xgb_shap_interaction_vals = model.predict(dtest, pred_interactions=True)
```

```
[37]: print(f"shap values shape: {xgb_shap_interaction_vals.shape}")
```

```
shap values shape: (2560, 13, 13)
```

Note that the expected value is again returned in the last column.

The `xgboost` library also implements an approximate feature attribution method, first described by Sabaas [here](#). This feature attribution method is similar in spirit to Shapley value, but does not account for the effect of variable order as explained [here](#) (pp. 10-11). This explanation method can be invoked as follows:

```
[38]: xgb_sabaas = model.predict(dtest, pred_contribs=True, approx_contribs=True)
```

Footnotes

(1): See the algorithm [overview](#) for a brief explanation of coverage.

(2): `model_output=raw` should always be used with the path-dependent perturbation for classification problems in `xgboost`, irrespective of whether the model is trained with the `binary:logitraw` or `binary:logistic`. Even though a model trained with the latter outputs probabilities, internally `xgboost` explains the output in margin space due to the `model_output=raw` option. To explain the probability output of a model, one should use the interventional algorithm and pass `model_output=probability` to the constructor along with the objective `binary:logistic` to the training function.

[source]

9.1 Measuring the linearity of machine learning models

9.1.1 Overview

Machine learning models include in general linear and non-linear operations: neural networks may include several layers consisting of linear algebra operations followed by non-linear activation functions, while models based on decision trees are by nature highly non-linear. The linearity measure function and class provide an operational definition for the amount of non-linearity of a map acting on vector spaces. Roughly speaking, the amount of non-linearity of the map is defined based on how much the output of the map applied to a linear superposition of input vectors differs from the linear superposition of the map's outputs for each individual vector. In the context of supervised learning, this definition is immediately applicable to machine learning models, which are fundamentally maps from a input vector space (the feature space) to an output vector space that may represent probabilities (for classification models) or actual values of quantities of interest (for regression models).

Given an input vector space V , an output vector space W and a map $M : V \rightarrow W$, the amount of non-linearity of the map M in a region β of the input space V and relative to some coefficients $\alpha(v)$ is defined as

$$L_{\beta,\alpha}^{(M)} = \left\| \int_{\beta} \alpha(v) M(v) dv - M \left(\int_{\beta} \alpha(v) v dv \right) \right\|,$$

where $v \in V$ and $\| \cdot \|$ denotes the norm of a vector. If we consider a finite number of vectors N , the amount of non-linearity can be defined as

$$L_{\beta,\alpha}^{(M)} = \left\| \sum_i \alpha_i M(v_i) - M \left(\sum_i \alpha_i v_i \right) \right\|,$$

where, with an abuse of notation, β is no longer a continuous region in the input space but a collection of input vectors $\{v_i\}$ and α is no longer a function but a collection of real coefficients $\{\alpha_i\}$ with $i \in \{1, \dots, N\}$. Note that the second expression may be interpreted as an approximation of the integral quantity defined in the first expression, where the vectors $\{v_i\}$ are sampled uniformly in the region β .

9.1.2 Application to machine learning models

In supervised learning, a model can be considered as a function M mapping vectors from the input space (feature vectors) to vectors in the output space. The output space may represent probabilities in the case of a classification model or values of the target quantities in the case of a regression model. The definition of the linearity measure given above can be applied to the case of a regression model (either a single target regression or a multi target regression) in a straightforward way.

In case of a classifier, let us denote by z the logits vector of the model such that the probabilities of the model M are given by $\text{softmax}(z)$. Since the activation function of the last layer is usually highly non-linear, it is convenient to apply the definition of linearity given above to the logits vector z . In the “white box” scenario, in which we have access to the internal architecture of the model, the vector z is accessible and the amount of non-linearity can be calculated immediately. On the other hand, if the only accessible quantities are the output probabilities (the “black box” scenario), we need to invert the last layer’s activation function in order to retrieve z . In other words, that means defining a new map $M' = f^{-1} \circ M(v)$ where f is the activation function at the last layer and considering $L_{\beta, \alpha}^{(M')}$ as a measure of the non-linearity of the model. The activation function of the last layer is usually a sigmoid function for binary classification tasks or a softmax function for multi-class classification. The inversion of the sigmoid function does not present any particular challenge, and the map M' can be written as

$$M' = -\log \circ \left(\frac{1 - M(v)}{M(v)} \right).$$

On the other hand, the softmax probabilities p are defined in terms of the vector z as $p_j = e^{z_j} / \sum_j e^{z_j}$, where z_j are the components of z . The inverse of the softmax function is thus defined up to a constant C which does not depend on j but might depend on the input vector v . The inverse map $M' = \text{softmax}^{-1} \circ M(v)$ is then given by:

$$M' = \log \circ M(v) + C(v),$$

where $C(v)$ is an arbitrary constant depending in general on the input vector v .

Since in the black box scenario it is not possible to assess the value of C , henceforth we will ignore it and define the amount of non-linearity of a machine learning model whose output is a probability distribution as

$$L_{\beta, \alpha}^{(\log \circ M)} = \left\| \sum_i \alpha_i \log \circ M(v_i) - \log \circ M \left(\sum_i \alpha_i v_i \right) \right\|.$$

It must be noted that the quantity above may in general be different from the “actual” amount of non-linearity of the model, i.e. the quantity calculated by accessing the activation vectors z directly.

9.1.3 Implementation

Sampling

The module implements two different methods for the sampling of vectors in a neighbourhood of the instance of interest v .

- The first sampling method **grid** consists of defining the region β as a discrete lattice of a given size around the instance of interest, with the size defined in terms of the L1 distance in the lattice; the vectors are then sampled from the lattice according to a uniform distribution. The density and the size of the lattice are controlled by the resolution parameter **res** and the size parameter **epsilon**. This method is highly efficient and scalable from a computational point of view.
- The second sampling method **knn** consists of sampling from the same probability distribution the instance v was drawn from; this method is implemented by simply selecting the K nearest neighbours to v from a training set, when this is available. The **knn** method imposes the constraint that the neighbourhood of v must include only vectors from the training set, and as a consequence it will exclude out-of-distribution instances from the computation of linearity.

Pairwise vs global linearity

The module implements two different methods to associate a value of the linearity measure to v .

- The first method consists of measuring the **global** linearity in a region around v . This means that we sample N vectors $\{v_i\}$ from a region β of the input space around v and apply

$$L_{\beta, \alpha}^{(M)} = \left\| \sum_{i=1}^N \alpha_i M(v_i) - M \left(\sum_{i=1}^N \alpha_i v_i \right) \right\|,$$

- The second method consists of measuring the **pairwise** linearity between the instance of interest and other vectors close to it, averaging over all such pairs. In other words, we sample N vectors $\{v_i\}$ from β as in the global method, but in this case we calculate the amount of non-linearity $L_{(v, v_i), \alpha}$ for every pair of vectors (v, v_i) and average over all the pairs. Given two coefficients $\{\alpha_0, \alpha_1\}$ such that $\alpha_0 + \alpha_1 = 1$, we can define the pairwise linearity measure relative to the instance of interest v as

$$L^{(M)} = \frac{1}{N} \sum_{i=0}^N \|\alpha_0 M(v) + \alpha_1 M(v_i) - M(\alpha_0 v + \alpha_1 v_i)\|.$$

The two methods are slightly different from a conceptual point of view: the global linearity measure combines all N vectors sampled in β in a single superposition, and can be conceptually regarded as a direct approximation of the integral quantity. Thus, the quantity is strongly linked to the model behavior in the whole region β . On the other hand, the pairwise linearity measure is an averaged quantity over pairs of superimposed vectors, with the instance of interest v included in each pair. For that reason, it is conceptually more tied to the instance v itself rather than the region β around it.

9.1.4 Usage

LinearityMeasure class

Given a `model` class with a `predict` method that return probabilities distribution in case of a classifier or numeric values in case of a regressor, the linearity measure L around an instance of interest X can be calculated using the class `LinearityMeasure` as follows:

```
from alibi.confidence import LinearityMeasure

predict_fn = lambda x: model.predict(x)

lm = LinearityMeasure(method='grid',
                      epsilon=0.04,
                      nb_samples=10,
                      res=100,
                      alphas=None,
                      model_type='classifier',
                      agg='pairwise',
                      verbose=False)

lm.fit(X_train)
L = lm.score(predict_fn, X)
```

Where `x_train` is the dataset the model was trained on. The `feature_range` is inferred from `x_train` in the `fit` step.

linearity_measure function

Given a `model` class with a `predict` method that return probabilities distribution in case of a classifier or numeric values in case of a regressor, the linearity measure L around an instance of interest X can also be calculated using the `linearity_measure` function as follows:

```
from alibi.confidence import linearity_measure
from alibi.confidence.model_linearity import infer_feature_range

predict_fn = lambda x: model.predict(x)

feature_range = infer_feature_range(X_train)
L = linearity_measure(predict_fn,
                      X,
                      feature_range=feature_range,
                      method='grid',
                      X_train=None,
                      epsilon=0.04,
                      nb_samples=10,
                      res=100,
                      alphas=None,
                      agg='global',
                      model_type='classifier')
```

Note that in this case the `feature_range` must be explicitly passed to the function and it is inferred beforehand.

9.1.5 Examples

Iris dataset

Fashion MNIST dataset

[source]

9.2 Trust Scores

9.2.1 Overview

It is important to know when a machine learning classifier's predictions can be trusted. Relying on the classifier's (uncalibrated) prediction probabilities is not optimal and can be improved upon. Enter *trust scores*. Trust scores measure the agreement between the classifier and a modified nearest neighbor classifier on the predicted instances. The trust score is the ratio between the distance of the instance to the nearest class different from the predicted class and the distance to the predicted class. A score of 1 would mean that the distance to the predicted class is the same as to the nearest other class. Higher scores correspond to more trustworthy predictions. The original paper on which the algorithm is based is called [To Trust Or Not To Trust A Classifier](#). Our implementation borrows heavily from and extends the authors' open source [code](#).

The method requires labeled training data to build *k-d trees* for each prediction class. When the classifier makes predictions on a test instance, we measure the distance of the instance to each of the trees. The trust score is then calculated by taking the ratio of the smallest distance to any other class than the predicted class and the distance to the predicted class. The distance is measured to the k th nearest neighbor in each tree or by using the average distance from the first to the k th neighbor.

In order to filter out the impact of outliers in the training data, they can optionally be removed using 2 filtering techniques. The first technique builds a k-d tree for each class and removes a fraction α of the training instances with the largest k nearest neighbor (kNN) distance to the other instances in the class. The second fits a kNN-classifier to the training set, and removes a fraction α of the training instances with the highest prediction class disagreement. Be aware that the first method operates on the prediction class level while the second method runs on the whole training set. It is also important to keep in mind that kNN methods might not be suitable when there are significant scale differences between the input features.

Trust scores can for instance be used as a warning flag for machine learning predictions. If the score drops below a certain value and there is disagreement between the model probabilities and the trust score, the prediction can be explained using techniques like anchors or contrastive explanations.

Trust scores work best for low to medium dimensional feature spaces. When working with high dimensional observations like images, dimensionality reduction methods (e.g. auto-encoders or PCA) could be applied as a pre-processing step before computing the scores. This is demonstrated by the following example [notebook](#).

9.2.2 Usage

Initialization and fit

At initialization, the optional filtering method used to remove outliers during the `fit` stage needs to be specified as well:

```
from alibi.confidence import TrustScore

ts = TrustScore(alpha=.05,
                filter_type='distance_knn',
                k_filter=10,
                leaf_size=40,
                metric='euclidean',
                dist_filter_type='point')
```

All the **hyperparameters** are optional:

- **alpha**: target fraction of instances to filter out.
- **filter_type**: filter method; one of *None* (no filtering), *distance_knn* (first technique discussed in *Overview*) or *probability_knn* (second technique).
- **k_filter**: number of neighbors used for the distance or probability based filtering method.
- **leaf_size**: affects the speed and memory usage to build the k-d trees. The memory scales with the ratio between the number of samples and the leaf size.
- **metric**: distance metric used for the k-d trees. *Euclidean* by default.
- **dist_filter_type**: *point* uses the distance to the *k*-nearest point while *mean* uses the average distance from the 1st to the *k*th nearest point during filtering.

In this example, we use the *distance_knn* method to filter out 5% of the instances of each class with the largest distance to its 10th nearest neighbor in that class:

```
ts.fit(X_train, y_train, classes=3)
```

- **classes**: equals the number of prediction classes.

X_train is the training set and *y_train* represents the training labels, either using one-hot encoding (OHE) or simple class labels.

Scores

The trust scores are simply calculated through the `score` method. `score` also returns the class labels of the closest not predicted class as a numpy array:

```
score, closest_class = ts.score(X_test,
                                y_pred,
                                k=2,
                                dist_type='point')
```

`y_pred` can again be represented using both OHE or via class labels.

- `k`: k th nearest neighbor used to compute distance to for each class.
- `dist_type`: similar to the filtering step, we can compute the distance to each class either to the k -th nearest point (*point*) or by using the average distance from the 1st to the k th nearest point (*mean*).

9.2.3 Examples

Trust Scores applied to Iris

Trust Scores applied to MNIST

EXAMPLES

10.1 Measuring the linearity of machine learning models

10.1.1 Linearity measure applied to fashion MNIST

General definition

The model linearity module in alibi provides metric to measure how linear an ML model is. Linearity is defined based on how much the linear superposition of the model's outputs differs from the output of the same linear superposition of the inputs.

Given N input vectors v_i , N real coefficients α_i and a predict function $M(v_i)$, the linearity of the predict function is defined as

$$L = \left\| \sum_i \alpha_i M(v_i) - M\left(\sum_i \alpha_i v_i\right) \right\| \quad \text{If } M \text{ is a regressor}$$

$$L = \left\| \sum_i \alpha_i \log \circ M(v_i) - \log \circ M\left(\sum_i \alpha_i v_i\right) \right\| \quad \text{If } M \text{ is a classifier}$$

Note that a lower value of L means that the model M is more linear.

Alibi implementation

- Based on the general definition above, alibi calculates the linearity of a model in the neighborhood of a given instance v_0 .

Fashion MNIST data set

- We train a convolutional neural network to classify the images in the fashion MNIST dataset.
- We investigate the correlation between the model's linearity associated to a certain instance and the class the instance belong to.
- We also calculate the linearity measure for each internal layer of the CNN and show how linearity propagates through the model.

```
[2]: import pandas as pd
import numpy as np
import matplotlib
%matplotlib inline
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
from time import time

import tensorflow as tf

from alibi.confidence import linearity_measure, LinearityMeasure
from alibi.confidence.model_linearity import infer_feature_range

from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input,
    → Activation
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import backend as K
```

Load data fashion mnist

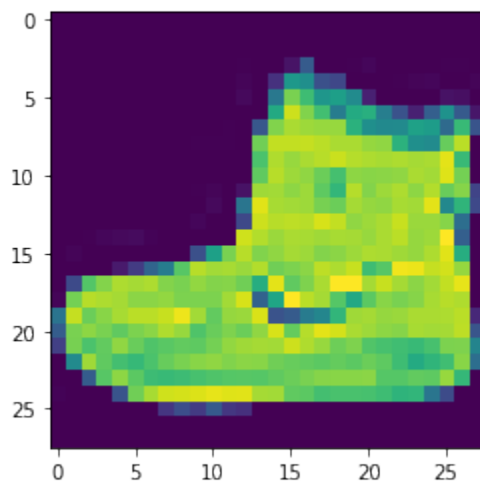
The fashion MNIST data set consists of 60000 images of shape 28×28 divided in 10 categories. Each category corresponds to a different type of clothing piece, such as “boots”, “t-shirts”, etc

```
[3]: (x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)
```

```
x_train shape: (60000, 28, 28) y_train shape: (60000,)
```

```
[4]: idx = 0
plt.imshow(x_train[idx])
print('Sample instance from the MNIST data set.')
```

```
Sample instance from the MNIST data set.
```



```
[5]: x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = np.reshape(x_train, x_train.shape + (1,))
x_test = np.reshape(x_test, x_test.shape + (1,))
print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
```

(continues on next page)

(continued from previous page)

```

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)

x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
y_train shape: (60000, 10) y_test shape: (10000, 10)

```

Convolutional neural network

Here we define and train a 2 layer convolutional neural network on the fashion MNIST data set.

Define model

```

[6]: def model():
    x_in = Input(shape=(28, 28, 1), name='input')
    x = Conv2D(filters=64, kernel_size=2, padding='same', name='conv_1')(x_in)
    x = Activation('relu', name='relu_1')(x)
    x = MaxPooling2D(pool_size=2, name='maxp_1')(x)
    x = Dropout(0.3, name='drop_1')(x)

    x = Conv2D(filters=64, kernel_size=2, padding='same', name='conv_2')(x)
    x = Activation('relu', name='relu_2')(x)
    x = MaxPooling2D(pool_size=2, name='maxp_2')(x)
    x = Dropout(0.3, name='drop_2')(x)

    x = Flatten(name='flat')(x)
    x = Dense(256, name='dense_1')(x)
    x = Activation('relu', name='relu_3')(x)
    x = Dropout(0.5, name='drop_3')(x)
    x_out = Dense(10, name='dense_2')(x)
    x_out = Activation('softmax', name='softmax')(x_out)

    cnn = Model(inputs=x_in, outputs=x_out)
    cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    return cnn

```

```

[7]: cnn = model()
     cnn.summary()

```

Model: "model"

Layer (type)	Output Shape	Param #
=====		
input (InputLayer)	[(None, 28, 28, 1)]	0

conv_1 (Conv2D)	(None, 28, 28, 64)	320

relu_1 (Activation)	(None, 28, 28, 64)	0

(continues on next page)

(continued from previous page)

maxp_1 (MaxPooling2D)	(None, 14, 14, 64)	0
drop_1 (Dropout)	(None, 14, 14, 64)	0
conv_2 (Conv2D)	(None, 14, 14, 64)	16448
relu_2 (Activation)	(None, 14, 14, 64)	0
maxp_2 (MaxPooling2D)	(None, 7, 7, 64)	0
drop_2 (Dropout)	(None, 7, 7, 64)	0
flat (Flatten)	(None, 3136)	0
dense_1 (Dense)	(None, 256)	803072
relu_3 (Activation)	(None, 256)	0
drop_3 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 10)	2570
softmax (Activation)	(None, 10)	0
=====		
Total params: 822,410		
Trainable params: 822,410		
Non-trainable params: 0		
=====		

Training

```
[8]: cnn.fit(x_train, y_train, batch_size=64, epochs=5);
```

```
Epoch 1/5
60000/60000 [=====] - 40s 674us/sample - loss: 0.5552 - acc: 0.
↪7955
Epoch 2/5
60000/60000 [=====] - 43s 717us/sample - loss: 0.3865 - acc: 0.
↪8596
Epoch 3/5
60000/60000 [=====] - 51s 852us/sample - loss: 0.3421 - acc: 0.
↪8765
Epoch 4/5
60000/60000 [=====] - 47s 782us/sample - loss: 0.3123 - acc: 0.
↪8851
Epoch 5/5
60000/60000 [=====] - 48s 802us/sample - loss: 0.2938 - acc: 0.
↪8936
```

Linearity of each Layer

Here we calculate the linearity of the model considering each layer as the output in turn. The values are averaged over 100 random instances sampled from the training set.

Extract layers

```
[9]: inp = cnn.input
outs = {l.name: l.output for l in cnn.layers}
predict_fns = {name: K.function([inp], [out]) for name, out in outs.items()}
```

Calculate linearity

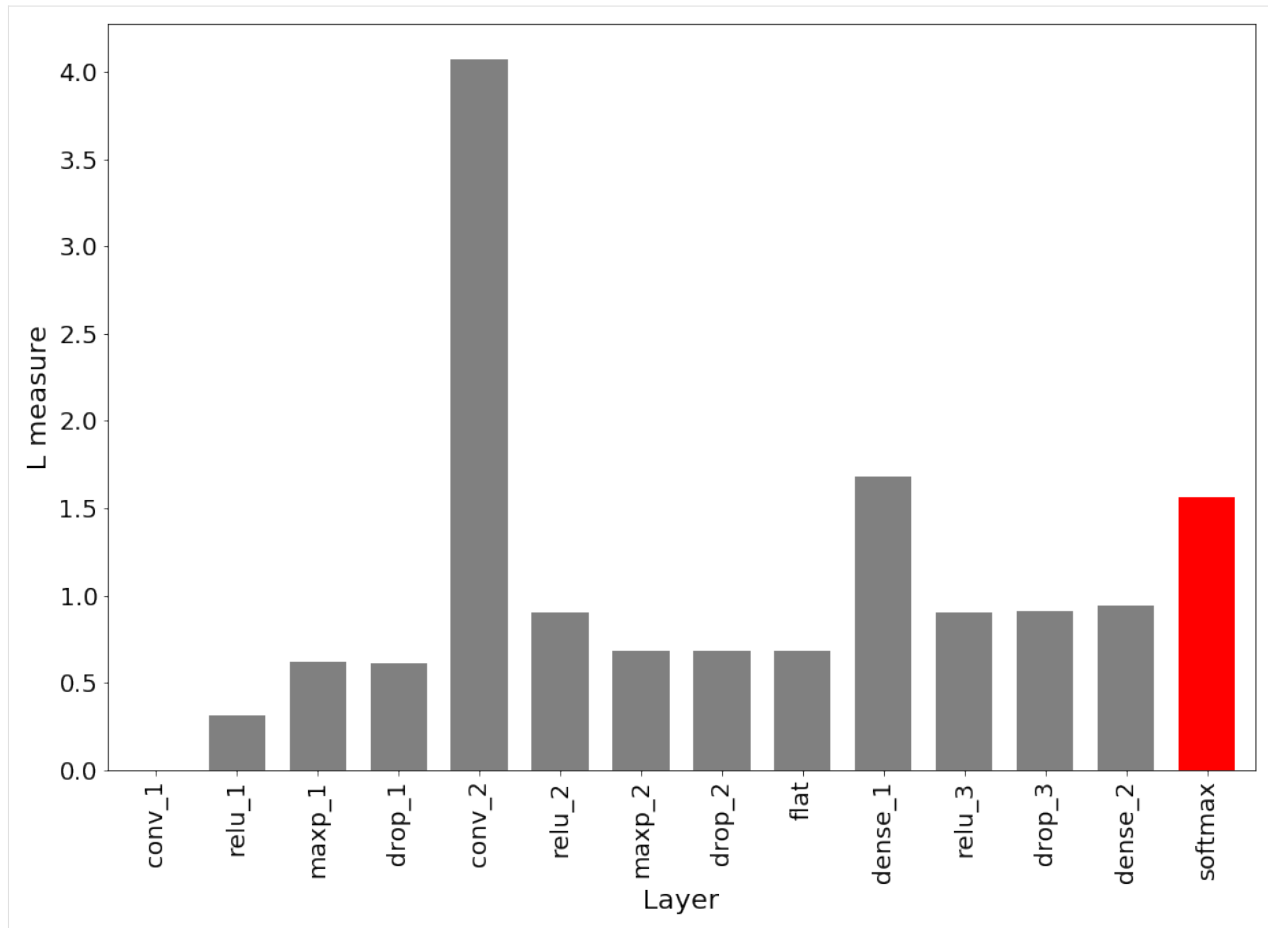
```
[10]: # Inferring feature ranges.
features_range = infer_feature_range(x_test)

# Selecting random instances from training set.
rnd = np.random.randint(len(x_test) - 101, size=100)

[11]: lins_layers = {}
for name, l in predict_fns.items():
    if name != 'input':
        def predict_fn(x):
            layer = l([x])
            return layer[0]
        if name == 'softmax':
            lins_layers[name] = linearity_measure(predict_fn, x_test[rnd], feature_
↳range=features_range,
                                                    agg='global', model_type='classifier',
↳nb_samples=20)
        else:
            lins_layers[name] = linearity_measure(predict_fn, x_test[rnd], feature_
↳range=features_range,
                                                    agg='global', model_type='regressor',
↳nb_samples=20)
lins_layers_mean = {k: v.mean() for k, v in lins_layers.items()}
S = pd.Series(data=lins_layers_mean)

[12]: colors = ['gray' for l in S[:-1]]
colors.append('r')
ax = S.plot(kind='bar', linewidth=3, figsize=(15,10), color=colors, width=0.7,
↳fontsize=18)
ax.set_ylabel('L measure', fontsize=20)
ax.set_xlabel('Layer', fontsize=20)
print('Linearity measure calculated taking as output each layer of a convolutional
↳neural network.')

Linearity measure calculated taking as output each layer of a convolutional neural
↳network.
```



Linearity measure in the locality of a given instance calculated taking as output each layer of a convolutional neural network trained on the fashion MNIST data set. * The linearity measure of the first convolutional layer conv_1 is 0, as expected since convolutions are linear operations. * The relu activation introduces non-linearity, which is increased by maxpooling. Dropout layers and flatten layers do not change the output at inference time so the linearity doesn't change. * The second convolutional layer conv_2 and the dense layers change the linearity even though they are linear operations. * The softmax layer in red is obtained by inverting the softmax function. * For more details see arxiv reference.

Linearity and categories

Here we calculate the linearity averaged over all instances belonging to the same class, for each class.

```
[13]: class_groups = []
      for i in range(10):
          y = y_test.argmax(axis=1)
          idxs_i = np.where(y == i)[0]
          class_groups.append(x_test[idxs_i])
```

```
[14]: def predict_fn(x):
      return cnn.predict(x)
      lins_classes = []
      t_0 = time()
```

(continues on next page)

(continued from previous page)

```

for j in range(len(class_groups)):
    print(f'Calculating linearity for instances belonging to class {j}')
    class_group = class_groups[j]
    class_group = np.random.permutation(class_group)[:2000]
    t_i = time()
    lin = linearity_measure(predict_fn, class_group, feature_range=features_range,
                           agg='global', model_type='classifier', nb_samples=20)
    t_i_1 = time() - t_i
    print(f'Run time for class {j}: {t_i_1}')
    lins_classes.append(lin)
t_fin = time() - t_0
print(f'Total run time: {t_fin}')

```

```

Calculating linearity for instances belonging to class 0
Run time for class 0: 2.941605806350708
Calculating linearity for instances belonging to class 1
Run time for class 1: 3.3313376903533936
Calculating linearity for instances belonging to class 2
Run time for class 2: 3.178601026535034
Calculating linearity for instances belonging to class 3
Run time for class 3: 3.324582815170288
Calculating linearity for instances belonging to class 4
Run time for class 4: 3.085338830947876
Calculating linearity for instances belonging to class 5
Run time for class 5: 3.159513473510742
Calculating linearity for instances belonging to class 6
Run time for class 6: 3.4014275074005127
Calculating linearity for instances belonging to class 7
Run time for class 7: 3.3238165378570557
Calculating linearity for instances belonging to class 8
Run time for class 8: 2.9885218143463135
Calculating linearity for instances belonging to class 9
Run time for class 9: 3.4760279655456543
Total run time: 32.22387504577637

```

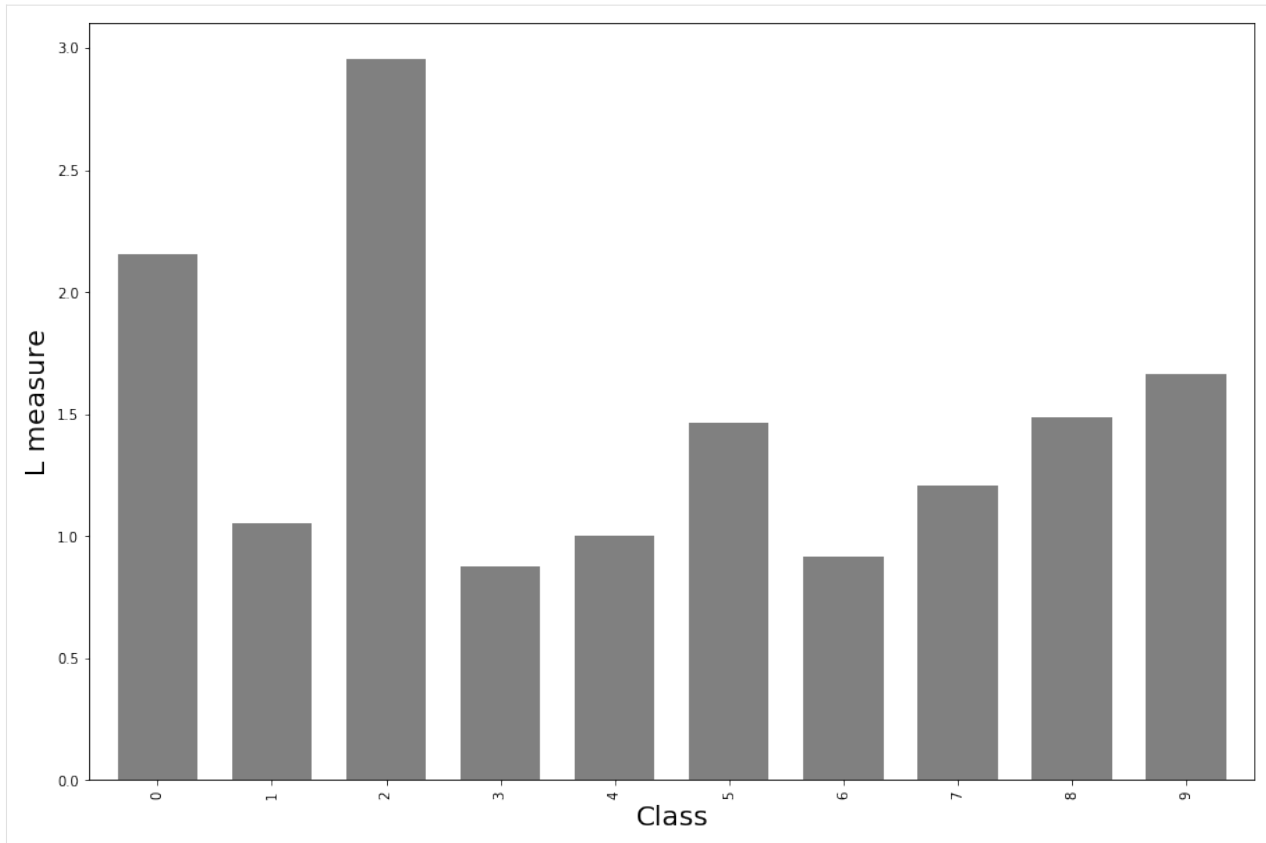
```
[15]: df = pd.DataFrame(data=lins_classes).T
```

```

[16]: ax = df.mean().plot(kind='bar', linewidth=3, figsize=(15,10), color='gray', width=0.7,
    ↪  fontsize=10)
    ax.set_ylabel('L measure', fontsize=20)
    ax.set_xlabel('Class', fontsize=20)
    print("Linearity measure distribution means for each class in the fashion MNIST data set.
    ↪ ")

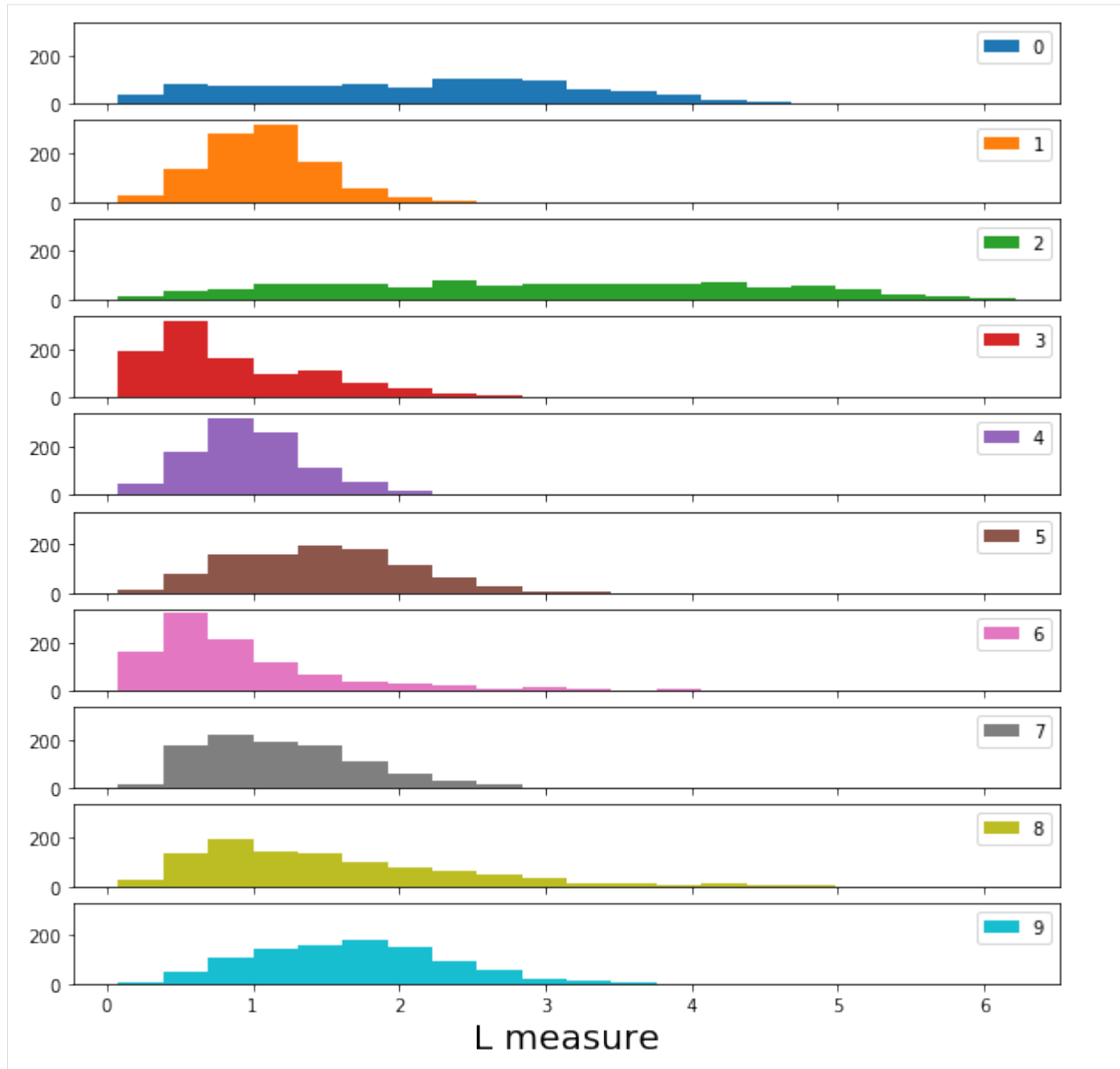
```

Linearity measure distribution means for each class in the fashion MNIST data set.



```
[17]: ax2 = df.plot(kind='hist', subplots=True, bins=20, figsize=(10,10), sharey=True)
      for a in ax2:
          a.set_xlabel('L measure', fontsize=20)
          a.set_ylabel('', rotation=True, fontsize=10)
      #ax2.set_ylabel('F', fontsize=10)
      print('Linearity measure distributions for each class in the fashion MNIST data set.')
```

Linearity measure distributions for each class in the fashion MNIST data set.



10.1.2 Linearity measure applied to Iris

General definition

The model linearity module in alibi provides metric to measure how linear an ML model is. Linearity is defined based on how much the linear superposition of the model's outputs differs from the output of the same linear superposition of the inputs.

Given N input vectors v_i , N real coefficients α_i and a predict function $M(v_i)$, the linearity of the predict function is defined as

$$L = \left\| \sum_i \alpha_i M(v_i) - M\left(\sum_i \alpha_i v_i\right) \right\| \quad \text{If } M \text{ is a regressor}$$

$$L = \left\| \sum_i \alpha_i \log \circ M(v_i) - \log \circ M\left(\sum_i \alpha_i v_i\right) \right\| \quad \text{If } M \text{ is a classifier}$$

Note that a lower value of L means that the model M is more linear.

Alibi implementation

- Based on the general definition above, alibi calculates the linearity of a model in the neighborhood of a given instance v_0 .

Iris Data set

- As an example, we will visualize the decision boundaries and the values of the linearity measure for various classifier on the iris dataset. Only 2 features are included for visualization purposes.

This example will use the `xgboost` library, which can be installed with:

```
[ ]: !pip install xgboost
```

```
[2]: import pandas as pd
import numpy as np
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from xgboost import XGBClassifier

from itertools import product
from alibi.confidence import linearity_measure, LinearityMeasure
```

Dataset

```
[3]: ds = load_iris()
X_train, y_train = ds.data[:, :2], ds.target
```

```
[4]: lins_dict = {}
```

Models

We will experiment with 5 different classifiers: * A logistic regression model, which is expected to be highly linear. * A random forest classifier, which is expected to be highly non-linear. * An xgboost classifier. * A support vector machine classifier. * A feed forward neural network

```
[5]: lr = LogisticRegression(fit_intercept=False, multi_class='multinomial', solver='newton-cg
↳ ')
    rf = RandomForestClassifier(n_estimators=100)
    xgb = XGBClassifier(n_estimators=100)
    svm = SVC(gamma=.1, kernel='rbf', probability=True)
    nn = MLPClassifier(hidden_layer_sizes=(100,50), activation='relu', max_iter=1000)
```

```
[6]: lr.fit(X_train, y_train)
    rf.fit(X_train, y_train)
    xgb.fit(X_train, y_train)
    svm.fit(X_train, y_train)
    nn.fit(X_train, y_train);
```

Decision boundaries and linearity

```
[7]: # Creating a grid
    x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
    y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
```

```
[8]: # Flattening points in the grid
    X = np.empty((len(xx.flatten()), 2))
    for i in range(xx.shape[0]):
        for j in range(xx.shape[1]):
            k = i * xx.shape[1] + j
            X[k] = np.array([xx[i, j], yy[i, j]])
```

Logistic regression

```
[9]: # Defining predict function for logistic regression
    clf = lr
    predict_fn = lambda x: clf.predict_proba(x)
```

```
[10]: # Calculating linearity for all points in the grid
    lm = LinearityMeasure(agg='pairwise')
    lm.fit(X_train)
    L = lm.score(predict_fn, X)
    L = L.reshape(xx.shape)
    lins_dict['LR'] = L.mean()
```

```
[11]: # Visualising decision boundaries and linearity values
    f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
    idx = (0,0)
```

(continues on next page)

(continued from previous page)

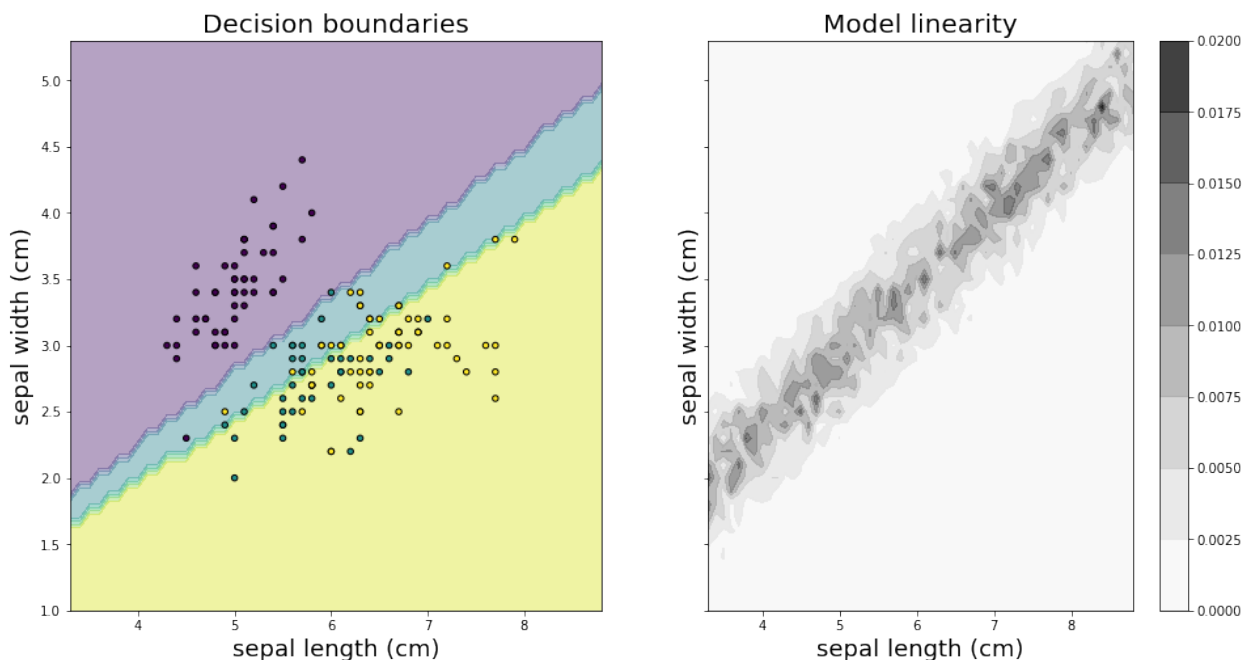
```

Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

axarr[0].contourf(xx, yy, Z, alpha=0.4)
axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k', alpha=1)
axarr[0].set_title('Decision boundaries', fontsize=20)
axarr[0].set_xlabel('sepal length (cm)', fontsize=18)
axarr[0].set_ylabel('sepal width (cm)', fontsize=18)

LPL = axarr[1].contourf(xx, yy, L, alpha=0.8, cmap='Greys')
axarr[1].set_title('Model linearity', fontsize=20)
axarr[1].set_xlabel('sepal length (cm)', fontsize=18)
axarr[1].set_ylabel('sepal width (cm)', fontsize=18)
cbar = f.colorbar(LPL)
#cbar.ax.set_ylabel('Linearity')
plt.show()
print('Decision boundaries (left panel) and linearity measure (right panel) for a
↳ logistic regression (LG) classifier in feature space. The x and y axis in the plots
↳ represent the sepal length and the sepal width, respectively. Different colours
↳ correspond to different predicted classes. The markers represents the data points in
↳ the training set.')
print('Maximum value model linearity: {}'.format(np.round(L.max(), 5)))
print(f'Minimum value model linearity: {np.round(L.min(), 5)}')

```



Decision boundaries (left panel) and linearity measure (right panel) for a logistic
↳ regression (LG) classifier in feature space. The x and y axis in the plots represent
↳ the sepal length and the sepal width, respectively. Different colours correspond to
↳ different predicted classes. The markers represents the data points in the training
↳ set.

Maximum value model linearity: 0.01841

Minimum value model linearity: 0.0

Random forest

```
[12]: # Defining predict function for random forest
      clf = rf
      predict_fn = lambda x: clf.predict_proba(x)

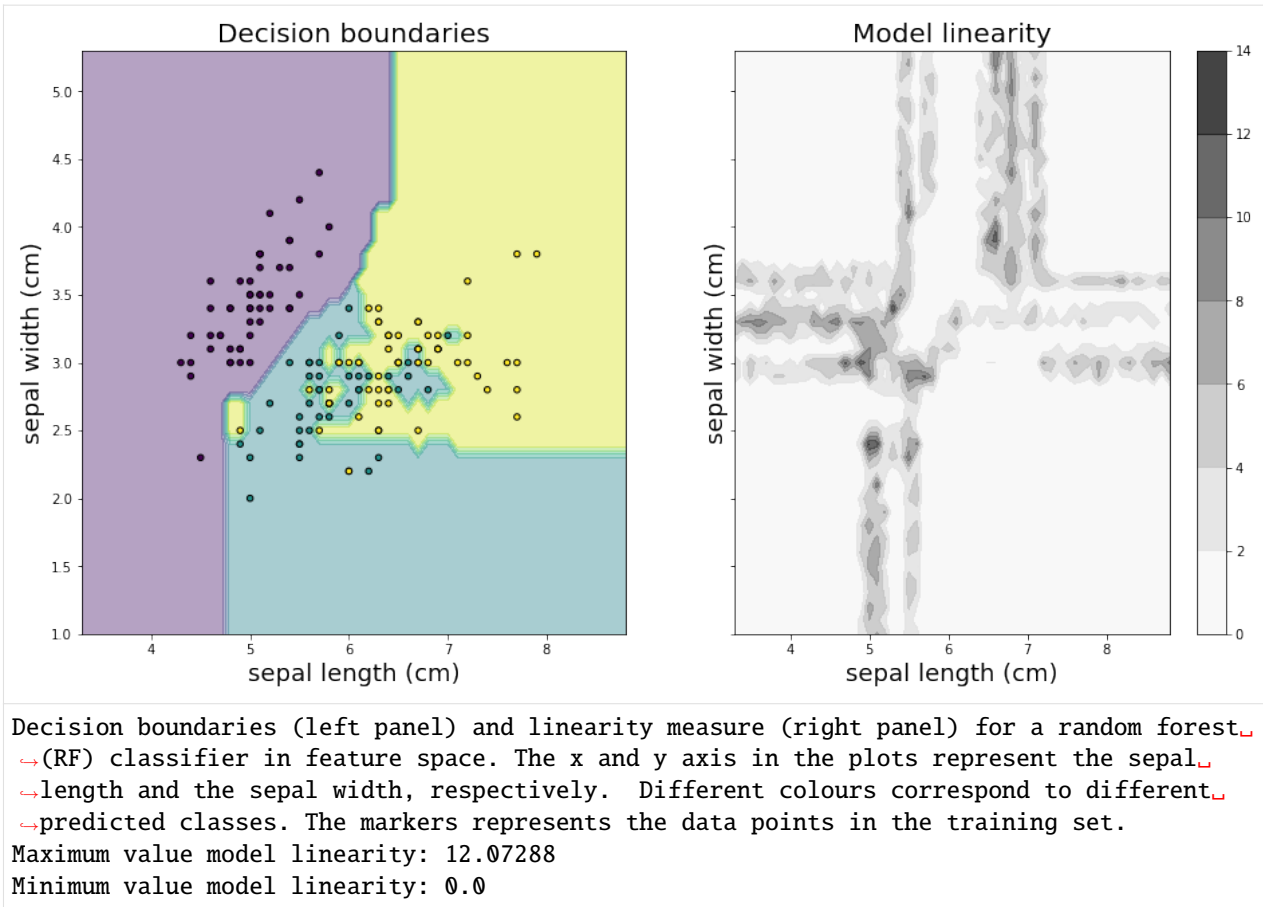
[13]: # Calculating linearity for all points in the grid
      lm = LinearityMeasure(agg='pairwise')
      lm.fit(X_train)
      L = lm.score(predict_fn, X)
      L = L.reshape(xx.shape)
      lins_dict['RF'] = L.mean()

[14]: # Visualising decision boundaries and linearity values
      f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
      idx = (0,0)
      Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
      Z = Z.reshape(xx.shape)

      axarr[0].contourf(xx, yy, Z, alpha=0.4)
      axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k', alpha=1)
      axarr[0].set_title('Decision boundaries', fontsize=20)
      axarr[0].set_xlabel('sepal length (cm)', fontsize=18)
      axarr[0].set_ylabel('sepal width (cm)', fontsize=18)

      LPL = axarr[1].contourf(xx, yy, L, alpha=0.8, cmap='Greys')
      axarr[1].set_title('Model linearity', fontsize=20)
      axarr[1].set_xlabel('sepal length (cm)', fontsize=18)
      axarr[1].set_ylabel('sepal width (cm)', fontsize=18)

      cbar = f.colorbar(LPL)
      plt.show()
      print('Decision boundaries (left panel) and linearity measure (right panel) for a random_
↳ forest (RF) classifier in feature space. The x and y axis in the plots represent the_
↳ sepal length and the sepal width, respectively. Different colours correspond to_
↳ different predicted classes. The markers represents the data points in the training_
↳ set.')
      print('Maximum value model linearity: {}'.format(np.round(L.max(), 5)))
      print(f'Minimum value model linearity: {np.round(L.min(), 5)}')
```



Xgboost

```
[15]: # Defining predict function for xgboost
      clf = xgb
      predict_fn = lambda x: clf.predict_proba(x)
```

```
[16]: # Calculating linearity for all points in the grid
      lm = LinearityMeasure(agg='pairwise')
      lm.fit(X_train)
      L = lm.score(predict_fn, X)
      L = L.reshape(xx.shape)
      lins_dict['XB'] = L.mean()
```

```
[17]: # Visualising decision boundaries and linearity values
      f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
      idx = (0, 0)
      Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
      Z = Z.reshape(xx.shape)

      axarr[0].contourf(xx, yy, Z, alpha=0.4)
      axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k', alpha=1)
```

(continues on next page)

(continued from previous page)

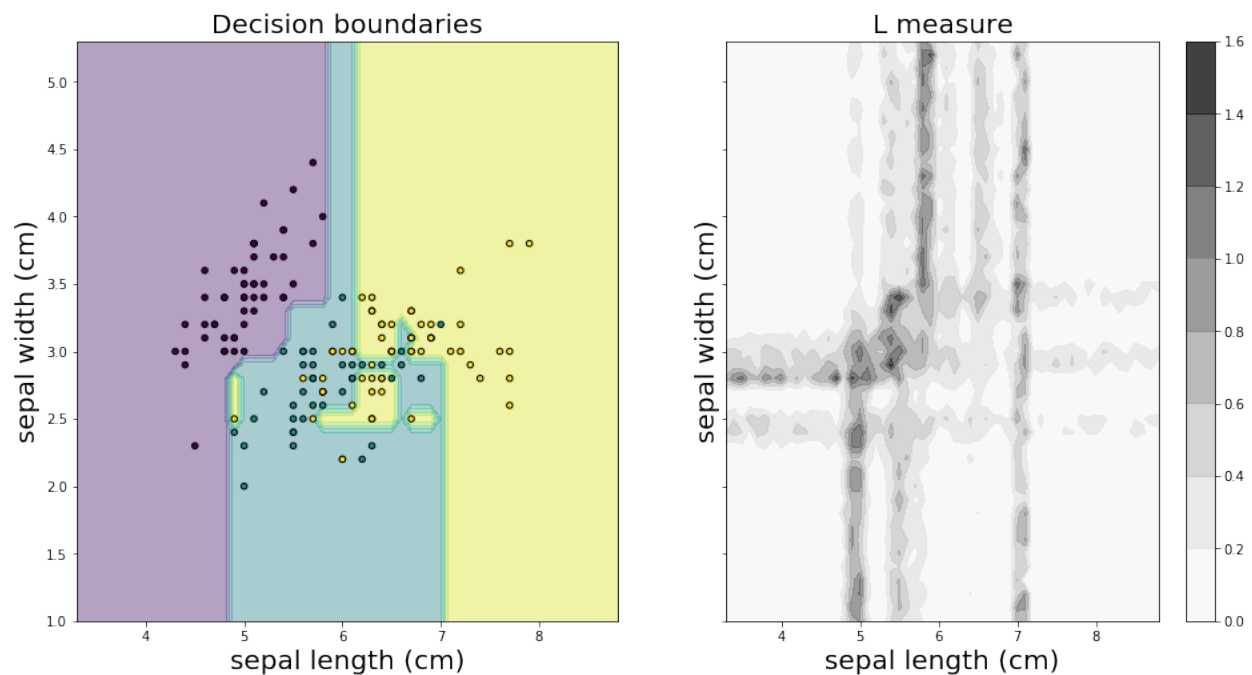
```

axarr[0].set_title('Decision boundaries', fontsize=20)
axarr[0].set_xlabel('sepal length (cm)', fontsize=20)
axarr[0].set_ylabel('sepal width (cm)', fontsize=20)

LPL = axarr[1].contourf(xx, yy, L, alpha=0.8, cmap='Greys')
axarr[1].set_title('L measure', fontsize=20)
axarr[1].set_xlabel('sepal length (cm)', fontsize=20)
axarr[1].set_ylabel('sepal width (cm)', fontsize=20)

cbar = f.colorbar(LPL)
#cbar.ax.set_ylabel('Linearity')
plt.show()
print('Decision boundaries (left panel) and linearity measure (right panel) for a
→ xgboost (XB) classifier in feature space. The x and y axis in the plots represent the
→ sepal length and the sepal width, respectively. Different colours correspond to
→ different predicted classes. The markers represents the data points in the training
→ set.')
print('Maximum value model linearity: {}'.format(np.round(L.max(), 5)))
print('Minimum value model linearity: {}'.format(np.round(L.min(), 5)))

```



Decision boundaries (left panel) and linearity measure (right panel) for a xgboost (XB) classifier in feature space. The x and y axis in the plots represent the sepal length and the sepal width, respectively. Different colours correspond to different predicted classes. The markers represents the data points in the training set.

Maximum value model linearity: 1.42648
Minimum value model linearity: 0.0

SVM

```
[18]: # Defining predict function for svm
      clf = svm
      predict_fn = lambda x: clf.predict_proba(x)

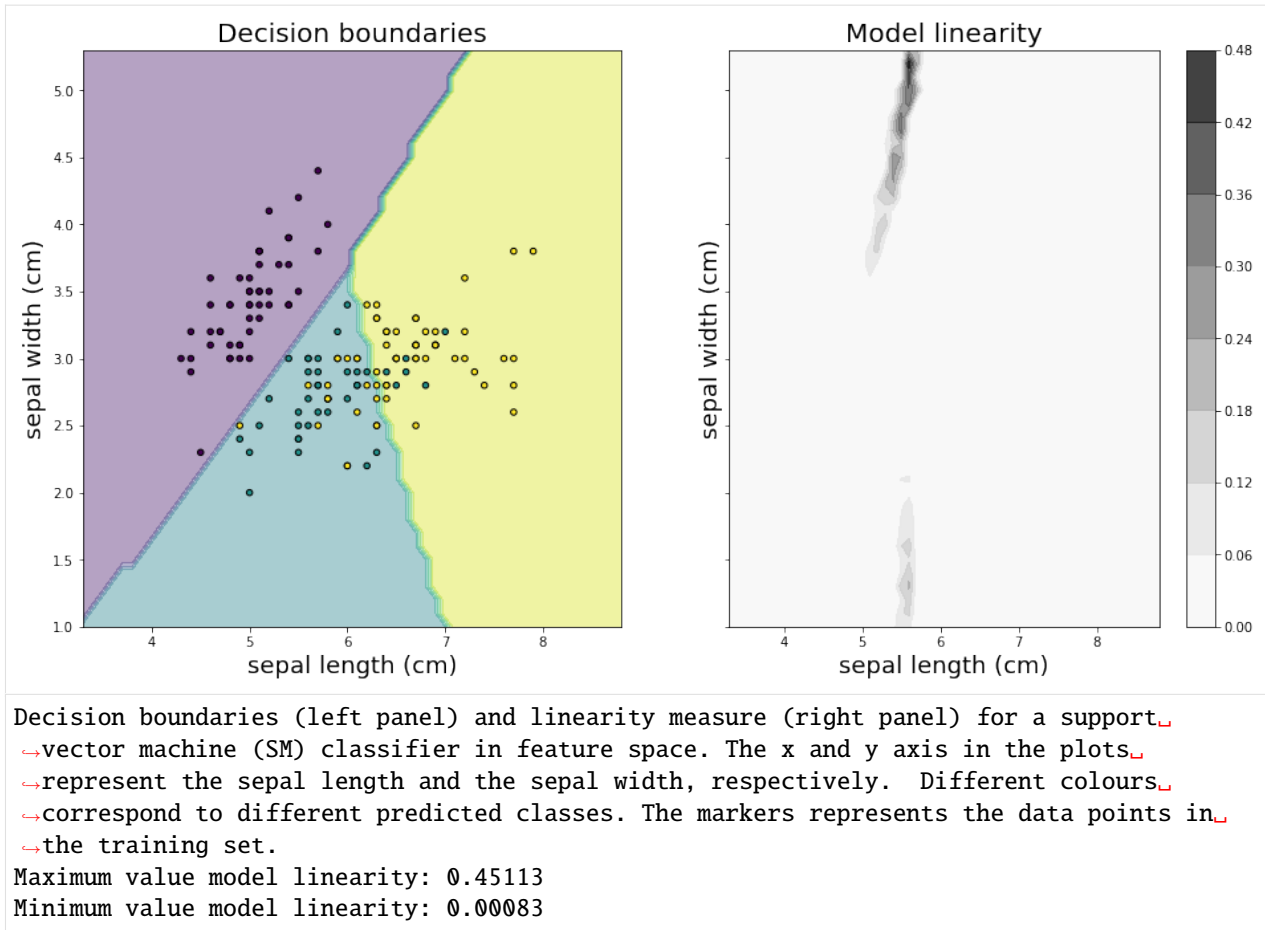
[19]: # Calculating linearity for all points in the grid
      lm = LinearityMeasure(agg='pairwise')
      lm.fit(X_train)
      L = lm.score(predict_fn, X)
      L = L.reshape(xx.shape)
      lins_dict['SM'] = L.mean()

[20]: # Visualising decision boundaries and linearity values
      f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
      idx = (0,0)
      Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
      Z = Z.reshape(xx.shape)

      axarr[0].contourf(xx, yy, Z, alpha=0.4)
      axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k', alpha=1)
      axarr[0].set_title('Decision boundaries', fontsize=20)
      axarr[0].set_xlabel('sepal length (cm)', fontsize=18)
      axarr[0].set_ylabel('sepal width (cm)', fontsize=18)

      LPL = axarr[1].contourf(xx, yy, L, alpha=0.8, cmap='Greys')
      axarr[1].set_title('Model linearity', fontsize=20)
      axarr[1].set_xlabel('sepal length (cm)', fontsize=18)
      axarr[1].set_ylabel('sepal width (cm)', fontsize=18)

      cbar = f.colorbar(LPL)
      #cbar.ax.set_ylabel('Linearity')
      plt.show()
      print('Decision boundaries (left panel) and linearity measure (right panel) for a_
      ↪support vector machine (SM) classifier in feature space. The x and y axis in the plots_
      ↪represent the sepal length and the sepal width, respectively. Different colours_
      ↪correspond to different predicted classes. The markers represents the data points in_
      ↪the training set.')
      print('Maximum value model linearity: {}'.format(np.round(L.max(), 5)))
      print(f'Minimum value model linearity: {np.round(L.min(), 5)}')
```



NN

```
[21]: # Defining predict function for svm
      clf = nn
      predict_fn = lambda x: clf.predict_proba(x)
```

```
[22]: # Calculating linearity for all points in the grid
      lm = LinearityMeasure(agg='pairwise')
      lm.fit(X_train)
      L = lm.score(predict_fn, X)
      L = L.reshape(xx.shape)
      lins_dict['NN'] = L.mean()
```

```
[23]: # Visualising decision boundaries and linearity values
      f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(16, 8))
      idx = (0,0)
      Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
      Z = Z.reshape(xx.shape)

      axarr[0].contourf(xx, yy, Z, alpha=0.4)
      axarr[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20, edgecolor='k', alpha=1)
```

(continues on next page)

(continued from previous page)

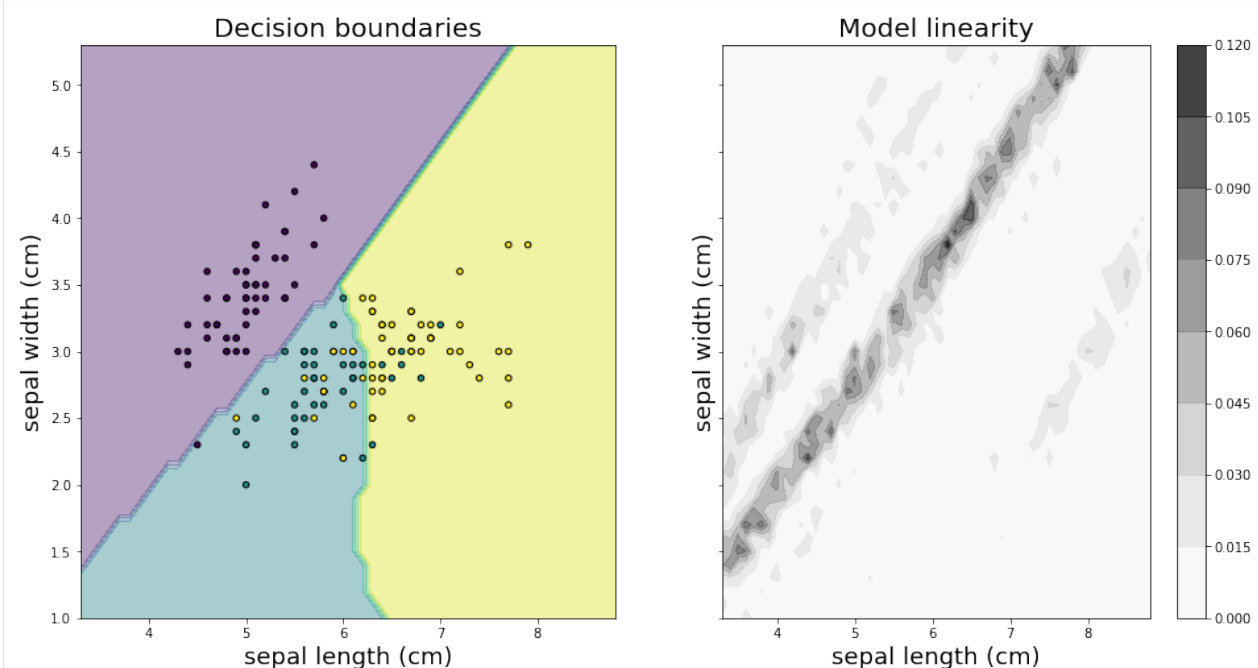
```

axarr[0].set_title('Decision boundaries', fontsize=20)
axarr[0].set_xlabel('sepal length (cm)', fontsize=18)
axarr[0].set_ylabel('sepal width (cm)', fontsize=18)

LPL = axarr[1].contourf(xx, yy, L, alpha=0.8, cmap='Greys')
axarr[1].set_title('Model linearity', fontsize=20)
axarr[1].set_xlabel('sepal length (cm)', fontsize=18)
axarr[1].set_ylabel('sepal width (cm)', fontsize=18)

cbar = f.colorbar(LPL)
#cbar.ax.set_ylabel('Linearity')
plt.show()
print('Decision boundaries (left panel) and linearity measure (right panel) for a feed_
↳ forward neural network classifier (NN) classifier in feature space. The x and y axis_
↳ in the plots represent the sepal length and the sepal width, respectively. Different_
↳ colours correspond to different predicted classes. The markers represents the data_
↳ points in the training set.')
print('Maximum value model linearity: {}'.format(np.round(L.max(), 5)))
print(f'Minimum value model linearity: {np.round(L.min(),5)}')

```



Decision boundaries (left panel) and linearity measure (right panel) for a feed forward_
↳ neural network classifier (NN) classifier in feature space. The x and y axis in the_
↳ plots represent the sepal length and the sepal width, respectively. Different colours_
↳ correspond to different predicted classes. The markers represents the data points in_
↳ the training set.

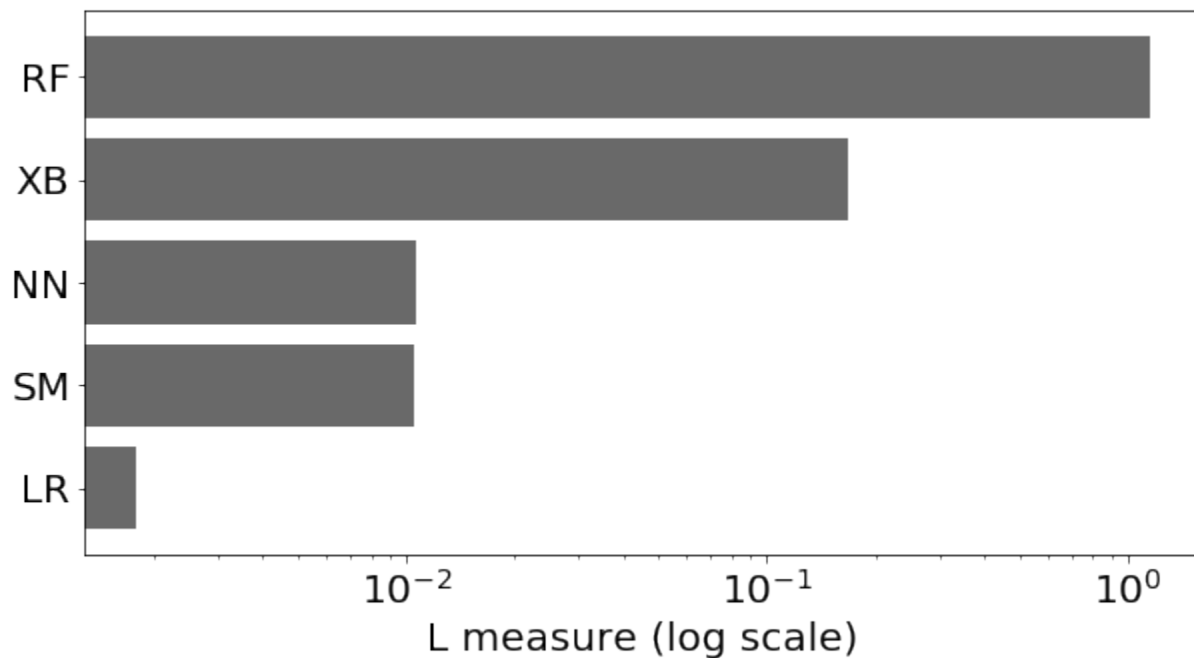
Maximum value model linearity: 0.11615

Minimum value model linearity: 3e-05

Average linearity over the whole feature space

```
[24]: ax = pd.Series(data=lins_dict).sort_values().plot(kind='barh', figsize=(10,5),
    ↪ fontsize=20, color='dimgray',
    ↪                                     width=0.8, logx=True)
ax.set_xlabel('L measure (log scale)', fontsize=20)
print('Comparison of the linearity measure L averaged over the whole feature space for
    ↪ various models trained on the iris dataset: random forest (RF), xgboost (XB), support
    ↪ vector machine (SM), neural network (NN) and logistic regression (LR). Note that the
    ↪ scale of the X axis is logarithmic.')
```

Comparison of the linearity measure L averaged over the whole feature space for various models trained on the iris dataset: random forest (RF), xgboost (XB), support vector machine (SM), neural network (NN) and logistic regression (LR). Note that the scale of the X axis is logarithmic.



10.2 Trust Scores

10.2.1 Trust Scores applied to Iris

It is important to know when a machine learning classifier's predictions can be trusted. Relying on the classifier's (uncalibrated) prediction probabilities is not optimal and can be improved upon. *Trust scores* measure the agreement between the classifier and a modified nearest neighbor classifier on the test set. The trust score is the ratio between the distance of the test instance to the nearest class different from the predicted class and the distance to the predicted class. Higher scores correspond to more trustworthy predictions. A score of 1 would mean that the distance to the predicted class is the same as to another class.

The original paper on which the algorithm is based is called [To Trust Or Not To Trust A Classifier](#). Our implementation borrows heavily from <https://github.com/google/TrustScore>, as does the example notebook.

```
[1]: import matplotlib
      %matplotlib inline
      import matplotlib.cm as cm
      import matplotlib.pyplot as plt
      import numpy as np
      import pandas as pd
      from sklearn.datasets import load_iris
      from sklearn.linear_model import LogisticRegression
      from sklearn.model_selection import StratifiedShuffleSplit
      from alibi.confidence import TrustScore
```

Load and prepare Iris dataset

```
[2]: dataset = load_iris()
```

Scale data

```
[3]: dataset.data = (dataset.data - dataset.data.mean(axis=0)) / dataset.data.std(axis=0)
```

Define training and test set

```
[4]: idx = 140
      X_train, y_train = dataset.data[:idx, :], dataset.target[:idx]
      X_test, y_test = dataset.data[idx+1:, :], dataset.target[idx+1:]
```

Fit model and make predictions

```
[5]: np.random.seed(0)
      clf = LogisticRegression(solver='liblinear', multi_class='auto')
      clf.fit(X_train, y_train)
      y_pred = clf.predict(X_test)
      print(f'Predicted class: {y_pred}')
      Predicted class: [2 2 2 2 2 2 2 2 2]
```

Basic Trust Score Usage

Initialise Trust Scores and fit on training data

The trust score algorithm builds *k-d trees* for each class. The distance of the test instance to the *k*th nearest neighbor of each tree (or the average distance to the *k*th neighbor) can then be used to calculate the trust score. We can optionally filter out outliers in the training data before building the trees. The example below uses the *distance_knn* (*filter_type*) method to filter out the 5% (*alpha*) instances of each class with the highest distance to its 10th nearest neighbor (*k_filter*) in that class.

```
[6]: ts = TrustScore(k_filter=10, # nb of neighbors used for kNN distance or probability to
      ↪ filter out outliers
      alpha=.05, # target fraction of instances to filter out
      filter_type='distance_knn', # filter method: None, 'distance_knn' or
```

(continues on next page)

(continued from previous page)

```

↪ 'probability_knn'
    leaf_size=40, # affects speed and memory to build KDTrees, memory_
↪ scales with n_samples / leaf_size
    metric='euclidean', # distance metric used for the KDTrees
    dist_filter_type='point') # 'point' uses distance to k-nearest point
                                # 'mean' uses average distance from the 1st to_
↪ the kth nearest point

```

```
[7]: ts.fit(X_train, y_train, classes=3) # classes = nb of prediction classes
```

Calculate Trust Scores on test data

Since the trust score is the ratio between the distance of the test instance to the nearest class different from the predicted class and the distance to the predicted class, higher scores correspond to more trustworthy predictions. A score of 1 would mean that the distance to the predicted class is the same as to another class. The `score` method returns arrays with both the trust scores and the class labels of the closest not predicted class.

```
[8]: score, closest_class = ts.score(X_test,
                                     y_pred, k=2, # kth nearest neighbor used
                                     # to compute distances for each class
                                     dist_type='point') # 'point' or 'mean' distance option

print(f'Trust scores: {score}')
print(f'\nClosest not predicted class: {closest_class}')
```

```

Trust scores: [2.574271277538439 2.1630334957870114 3.1629405367742223
 2.7258494544157927 2.541748027539072 1.402878283257114 1.941073062524019
 2.0601725424359296 2.1781121494573514]

```

```
Closest not predicted class: [1 1 1 1 1 1 1 1 1]
```

Comparison of Trust Scores with model prediction probabilities

Let's compare the prediction probabilities from the classifier with the trust scores for each prediction. The first use case checks whether trust scores are better than the model's prediction probabilities at identifying correctly classified examples, while the second use case does the same for incorrectly classified instances.

First we need to set up a couple of helper functions.

- Define a function that handles model training and predictions for a simple logistic regression:

```
[9]: def run_lr(X_train, y_train, X_test):
    clf = LogisticRegression(solver='liblinear', multi_class='auto')
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    y_pred_proba = clf.predict_proba(X_test)
    probas = y_pred_proba[range(len(y_pred)), y_pred] # probabilities of predicted class
    return y_pred, probas
```

- Define the function that generates the precision plots:

```
[10]: def plot_precision_curve(plot_title,
                               percentiles,
                               labels,
                               final_tp,
                               final_stderr,
                               final_misclassification,
                               colors = ['blue', 'darkorange', 'brown', 'red', 'purple']):

    plt.title(plot_title, fontsize=18)
    colors = colors + list(cm.rainbow(np.linspace(0, 1, len(final_tp))))
    plt.xlabel("Percentile", fontsize=14)
    plt.ylabel("Precision", fontsize=14)

    for i, label in enumerate(labels):
        ls = "--" if ("Model" in label) else "-"
        plt.plot(percentiles, final_tp[i], ls, c=colors[i], label=label)
        plt.fill_between(percentiles,
                         final_tp[i] - final_stderr[i],
                         final_tp[i] + final_stderr[i],
                         color=colors[i],
                         alpha=.1)

    if 0. in percentiles:
        plt.legend(loc="lower right", fontsize=14)
    else:
        plt.legend(loc="upper left", fontsize=14)
    model_acc = 100 * (1 - final_misclassification)
    plt.axvline(x=model_acc, linestyle="dotted", color="black")
    plt.show()
```

- The function below trains the model on a number of folds, makes predictions, calculates the trust scores, and generates the precision curves to compare the trust scores with the model prediction probabilities:

```
[11]: def run_precision_plt(X, y, nfolds, percentiles, run_model, test_size=.5,
                             plt_title="", plt_names=[], predict_correct=True, classes=3):

    def stderr(L):
        return np.std(L) / np.sqrt(len(L))

    all_tp = [[[ for p in percentiles] for _ in plt_names]
    misclassifications = []
    mult = 1 if predict_correct else -1

    folds = StratifiedShuffleSplit(n_splits=nfolds, test_size=test_size, random_state=0)
    for train_idx, test_idx in folds.split(X, y):
        # create train and test folds, train model and make predictions
        X_train, y_train = X[train_idx, :], y[train_idx]
        X_test, y_test = X[test_idx, :], y[test_idx]
        y_pred, probas = run_model(X_train, y_train, X_test)
        # target points are the correctly classified points
        target_points = np.where(y_pred == y_test)[0] if predict_correct else np.where(y_
↪pred != y_test)[0]
        final_curves = [probas]
```

(continues on next page)

(continued from previous page)

```

# calculate trust scores
ts = TrustScore()
ts.fit(X_train, y_train, classes=classes)
scores, _ = ts.score(X_test, y_pred)
final_curves.append(scores) # contains prediction probabilities and trust scores
# check where prediction probabilities and trust scores are above a certain
↪percentage level
    for p, perc in enumerate(percentiles):
        high_proba = [np.where(mult * curve >= np.percentile(mult * curve, perc))[0]]
↪for curve in final_curves
        if 0 in map(len, high_proba):
            continue
        # calculate fraction of values above percentage level that are correctly (or
↪incorrectly) classified
        tp = [len(np.intersect1d(hp, target_points)) / (1. * len(hp)) for hp in high_
↪proba]
        for i in range(len(plt_names)):
            all_tp[i][p].append(tp[i]) # for each percentile, store fraction of
↪values above cutoff value
        misclassifications.append(len(target_points) / (1. * len(X_test)))

# average over folds for each percentile
final_tp = [[] for _ in plt_names]
final_stderr = [[] for _ in plt_names]
for p, perc in enumerate(percentiles):
    for i in range(len(plt_names)):
        final_tp[i].append(np.mean(all_tp[i][p]))
        final_stderr[i].append(stderr(all_tp[i][p]))

for i in range(len(all_tp)):
    final_tp[i] = np.array(final_tp[i])
    final_stderr[i] = np.array(final_stderr[i])

final_misclassification = np.mean(misclassifications)

# create plot
plot_precision_curve(plt_title, percentiles, plt_names, final_tp, final_stderr,
↪final_misclassification)

```

Detect correctly classified examples

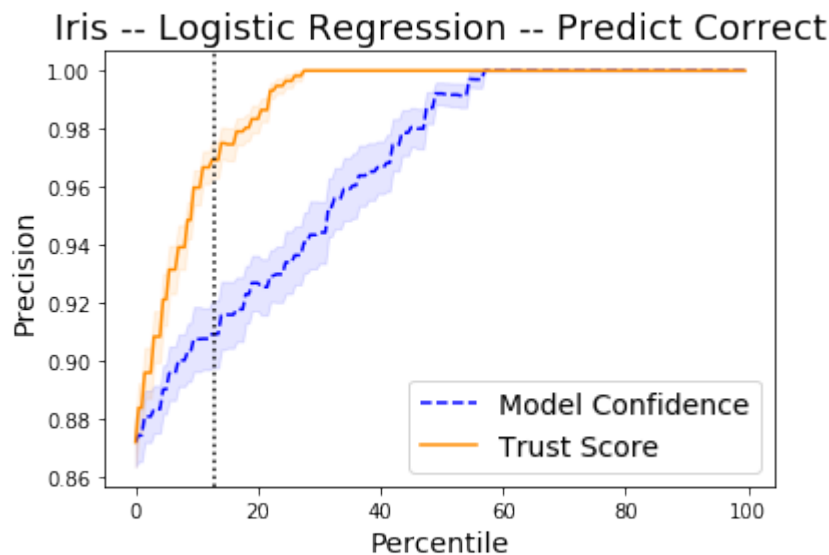
The x-axis on the plot below shows the percentiles for the model prediction probabilities of the predicted class for each instance and for the trust scores. The y-axis represents the precision for each percentile. For each percentile level, we take the test examples whose trust score is above that percentile level and plot the percentage of those points that were correctly classified by the classifier. We do the same with the classifier's own model confidence (i.e. softmax probabilities). For example, at percentile level 80, we take the top 20% scoring test examples based on the trust score and plot the percentage of those points that were correctly classified. We also plot the top 20% scoring test examples based on model probabilities and plot the percentage of those that were correctly classified. The vertical dotted line is the error of the logistic regression classifier. The plots are an average over 10 folds of the dataset with 50% of the data kept for the test set.

The *Trust Score* and *Model Confidence* curves then show that the model precision is typically higher when using the

trust scores to rank the predictions compared to the model prediction probabilities.

```
[12]: X = dataset.data
      y = dataset.target
      percentiles = [0 + 0.5 * i for i in range(200)]
      nolds = 10
      plt_names = ['Model Confidence', 'Trust Score']
      plt_title = 'Iris -- Logistic Regression -- Predict Correct'
```

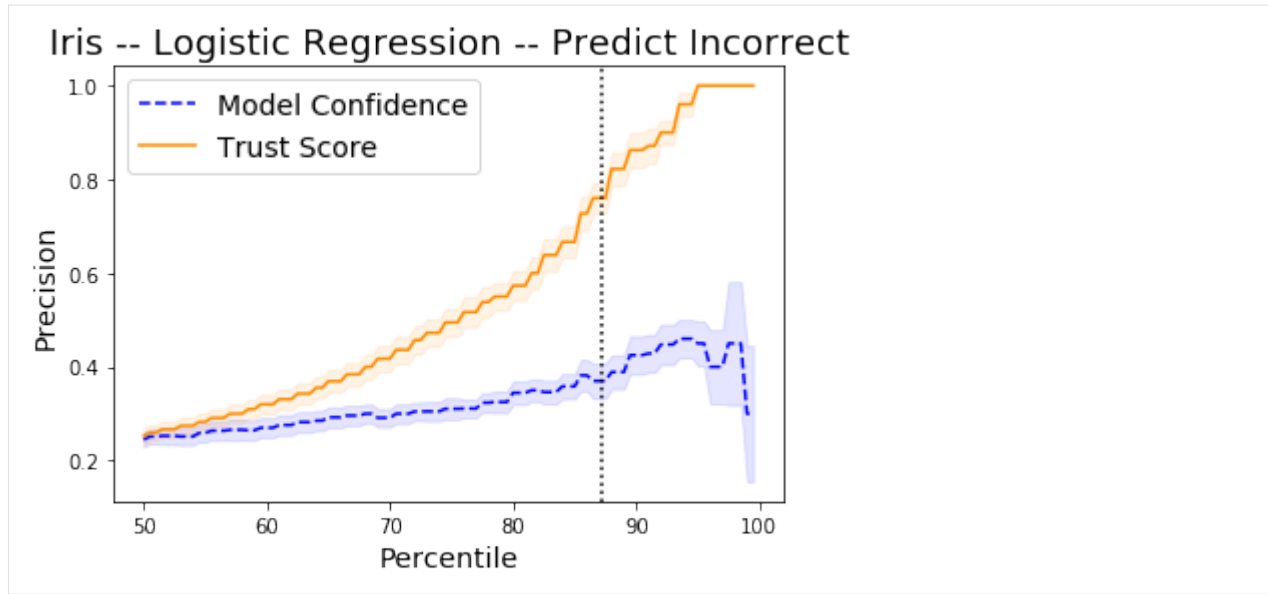
```
[13]: run_precision_plt(X, y, nolds, percentiles, run_lr, plt_title=plt_title,
                       plt_names=plt_names, predict_correct=True)
```



Detect incorrectly classified examples

By taking the *negative of the prediction probabilities and trust scores*, we can also see on the plot below how the trust scores compare to the model predictions for incorrectly classified instances. The vertical dotted line is the accuracy of the logistic regression classifier. The plot shows the precision of identifying incorrectly classified instances. Higher is obviously better.

```
[14]: percentiles = [50 + 0.5 * i for i in range(100)]
      plt_title = 'Iris -- Logistic Regression -- Predict Incorrect'
      run_precision_plt(X, y, nolds, percentiles, run_lr, plt_title=plt_title,
                       plt_names=plt_names, predict_correct=False)
```



10.2.2 Trust Scores applied to MNIST

It is important to know when a machine learning classifier's predictions can be trusted. Relying on the classifier's (uncalibrated) prediction probabilities is not optimal and can be improved upon. *Trust scores* measure the agreement between the classifier and a modified nearest neighbor classifier on the test set. The trust score is the ratio between the distance of the test instance to the nearest class different from the predicted class and the distance to the predicted class. Higher scores correspond to more trustworthy predictions. A score of 1 would mean that the distance to the predicted class is the same as to another class.

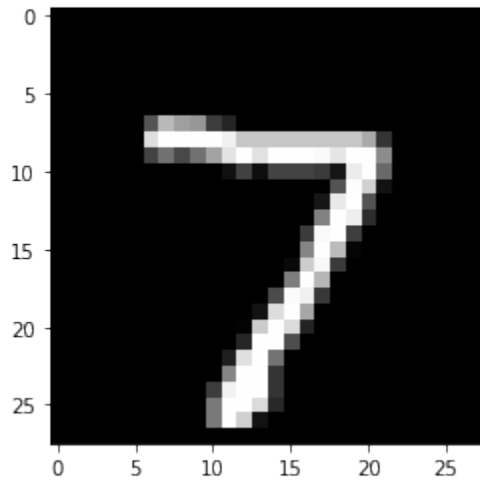
The original paper on which the algorithm is based is called [To Trust Or Not To Trust A Classifier](#). Our implementation borrows heavily from <https://github.com/google/TrustScore>, as does the example notebook.

Trust scores work best for low to medium dimensional feature spaces. This notebook illustrates how you can **apply trust scores to high dimensional** data like images by adding an additional pre-processing step in the form of an [auto-encoder](#) to reduce the dimensionality. Other dimension reduction techniques like PCA can be used as well.

```
[1]: import tensorflow as tf
from tensorflow.keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input,
    ↳ UpSampling2D
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.utils import to_categorical
import matplotlib
%matplotlib inline
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import StratifiedShuffleSplit
from alibi.confidence import TrustScore
```

```
[2]: (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
print('x_train shape:', x_train.shape, 'y_train shape:', y_train.shape)
plt.gray()
plt.imshow(x_test[0]);
```

```
x_train shape: (60000, 28, 28) y_train shape: (60000,)
```



Prepare data: scale, reshape and categorize

```
[3]: x_train = x_train.astype('float32') / 255
      x_test = x_test.astype('float32') / 255
      x_train = np.reshape(x_train, x_train.shape + (1,))
      x_test = np.reshape(x_test, x_test.shape + (1,))
      print('x_train shape:', x_train.shape, 'x_test shape:', x_test.shape)
      y_train = to_categorical(y_train)
      y_test = to_categorical(y_test)
      print('y_train shape:', y_train.shape, 'y_test shape:', y_test.shape)

      x_train shape: (60000, 28, 28, 1) x_test shape: (10000, 28, 28, 1)
      y_train shape: (60000, 10) y_test shape: (10000, 10)
```

```
[4]: xmin, xmax = -.5, .5
      x_train = ((x_train - x_train.min()) / (x_train.max() - x_train.min())) * (xmax - xmin) + xmin
      x_test = ((x_test - x_test.min()) / (x_test.max() - x_test.min())) * (xmax - xmin) + xmin
```

Define and train model

For this example we are not interested in optimizing model performance so a simple softmax classifier will do:

```
[5]: def sc_model():
      x_in = Input(shape=(28, 28, 1))
      x = Flatten()(x_in)
      x_out = Dense(10, activation='softmax')(x)
      sc = Model(inputs=x_in, outputs=x_out)
      sc.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
      return sc
```

```
[6]: sc = sc_model()
      sc.summary()
      sc.fit(x_train, y_train, batch_size=128, epochs=5);
```

Model: "model"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 10)	7850

Total params: 7,850

Trainable params: 7,850

Non-trainable params: 0

Epoch 1/5

60000/60000 [=====] - 1s 12us/sample - loss: 1.2706 - acc: 0.
 ↳ 6963

Epoch 2/5

60000/60000 [=====] - 1s 9us/sample - loss: 0.7030 - acc: 0.8422

Epoch 3/5

60000/60000 [=====] - 1s 9us/sample - loss: 0.5762 - acc: 0.8618

Epoch 4/5

60000/60000 [=====] - 1s 9us/sample - loss: 0.5155 - acc: 0.8706

Epoch 5/5

60000/60000 [=====] - 1s 9us/sample - loss: 0.4787 - acc: 0.8759

Evaluate the model on the test set:

```
[7]: score = sc.evaluate(x_test, y_test, verbose=0)
      print('Test accuracy: ', score[1])
```

Test accuracy: 0.8862

Define and train auto-encoder

```
[8]: def ae_model():
      # encoder
      x_in = Input(shape=(28, 28, 1))
      x = Conv2D(16, (3, 3), activation='relu', padding='same')(x_in)
      x = MaxPooling2D((2, 2), padding='same')(x)
      x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
      x = MaxPooling2D((2, 2), padding='same')(x)
      x = Conv2D(4, (3, 3), activation=None, padding='same')(x)
      encoded = MaxPooling2D((2, 2), padding='same')(x)
      encoder = Model(x_in, encoded)

      # decoder
      dec_in = Input(shape=(4, 4, 4))
      x = Conv2D(4, (3, 3), activation='relu', padding='same')(dec_in)
      x = UpSampling2D((2, 2))(x)
      x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
      x = UpSampling2D((2, 2))(x)
```

(continues on next page)

(continued from previous page)

```

x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation=None, padding='same')(x)
decoder = Model(dec_in, decoded)

# autoencoder = encoder + decoder
x_out = decoder(encoder(x_in))
autoencoder = Model(x_in, x_out)
autoencoder.compile(optimizer='adam', loss='mse')

return autoencoder, encoder, decoder

```

```

[9]: ae, enc, dec = ae_model()
ae.summary()
ae.fit(x_train, x_test, batch_size=128, epochs=8, validation_data=(x_test, x_test))
ae.save('mnist_ae.h5')
enc.save('mnist_enc.h5')

```

Model: "model_3"

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 28, 28, 1)]	0
model_1 (Model)	(None, 4, 4, 4)	1612
model_2 (Model)	(None, 28, 28, 1)	1757

Total params: 3,369
 Trainable params: 3,369
 Non-trainable params: 0

Train on 60000 samples, validate on 10000 samples

```

Epoch 1/8
60000/60000 [=====] - 29s 477us/sample - loss: 0.0606 - val_
↳ loss: 0.0399
Epoch 2/8
60000/60000 [=====] - 34s 572us/sample - loss: 0.0341 - val_
↳ loss: 0.0301
Epoch 3/8
60000/60000 [=====] - 43s 715us/sample - loss: 0.0288 - val_
↳ loss: 0.0272
Epoch 4/8
60000/60000 [=====] - 48s 806us/sample - loss: 0.0265 - val_
↳ loss: 0.0253
Epoch 5/8
60000/60000 [=====] - 41s 680us/sample - loss: 0.0249 - val_
↳ loss: 0.0239
Epoch 6/8
60000/60000 [=====] - 39s 649us/sample - loss: 0.0237 - val_
↳ loss: 0.0230
Epoch 7/8

```

(continues on next page)

(continued from previous page)

```
60000/60000 [=====] - 33s 545us/sample - loss: 0.0229 - val_
↳ loss: 0.0222
Epoch 8/8
60000/60000 [=====] - 29s 484us/sample - loss: 0.0224 - val_
↳ loss: 0.0217
```

```
[10]: ae = load_model('mnist_ae.h5')
      enc = load_model('mnist_enc.h5')
```

Calculate Trust Scores

Initialize trust scores:

```
[11]: ts = TrustScore()
```

The key is to **fit and calculate the trust scores on the encoded instances**. The encoded data still needs to be reshaped from (60000, 4, 4, 4) to (60000, 64) to comply with the k-d tree format. This is handled internally:

```
[12]: x_train_enc = enc.predict(x_train)
      ts.fit(x_train_enc, y_train, classes=10) # 10 classes present in MNIST
```

Reshaping data from (60000, 4, 4, 4) to (60000, 64) so k-d trees can be built.

We can now calculate the trust scores and closest not predicted classes of the predictions on the test set, using the distance to the 5th nearest neighbor in each class:

```
[13]: n_samples = 1000 # calculate the trust scores for the first 1000 predictions on the test_
      ↳ set
      x_test_enc = enc.predict(x_test[:n_samples])
      y_pred = sc.predict(x_test[:n_samples])
      score, closest_class = ts.score(x_test_enc[:n_samples], y_pred, k=5)
```

Reshaping data from (1000, 4, 4, 4) to (1000, 64) so k-d trees can be queried.

Let's inspect which predictions have low and high trust scores:

```
[14]: n = 5

# lowest and highest trust scores
idx_min, idx_max = np.argsort(score)[:n], np.argsort(score)[-n:]
score_min, score_max = score[idx_min], score[idx_max]
closest_min, closest_max = closest_class[idx_min], closest_class[idx_max]
pred_min, pred_max = y_pred[idx_min], y_pred[idx_max]
imgs_min, imgs_max = x_test[idx_min], x_test[idx_max]
label_min, label_max = np.argmax(y_test[idx_min], axis=1), np.argmax(y_test[idx_max],
↳ axis=1)

# model confidence percentiles
max_proba = y_pred.max(axis=1)

# low score high confidence examples
idx_low = np.where((max_proba>0.80) & (max_proba<0.9) & (score<1))[0][:n]
```

(continues on next page)

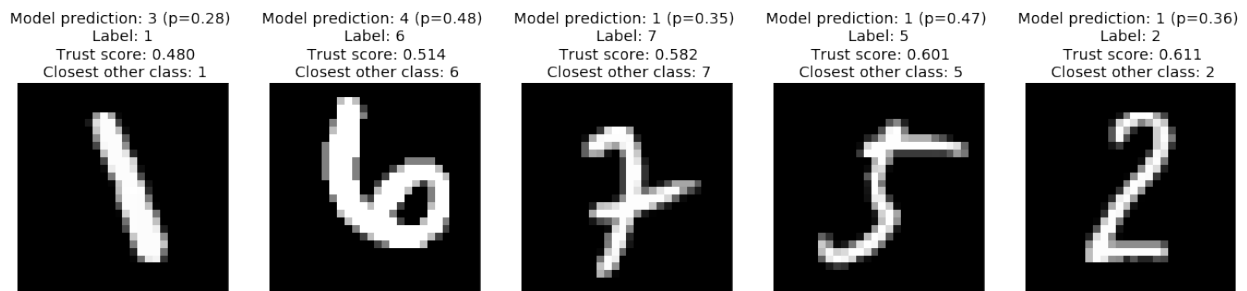
(continued from previous page)

```
score_low = score[idx_low]
closest_low = closest_class[idx_low]
pred_low = y_pred[idx_low]
imgs_low = x_test[idx_low]
label_low = np.argmax(y_test[idx_low], axis=1)
```

Low Trust Scores

The image below makes clear that the low trust scores correspond to misclassified images. Because the trust scores are significantly below 1, they correctly identified that the images belong to another class than the predicted class, and identified that class.

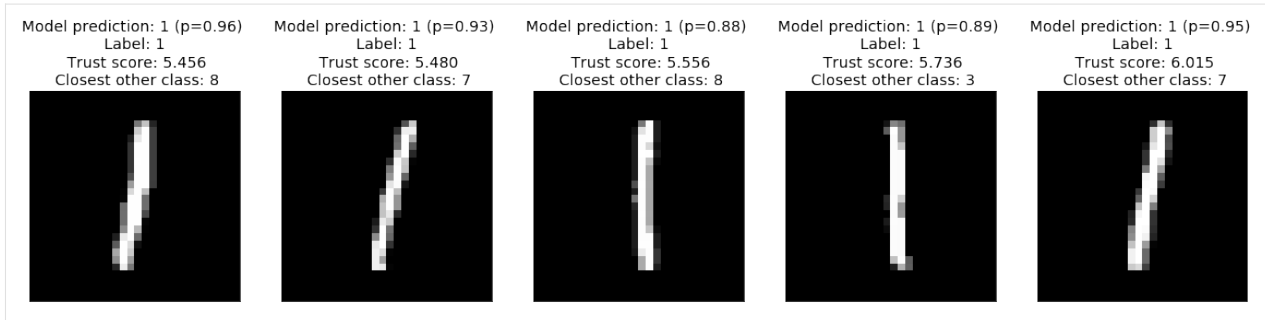
```
[15]: plt.figure(figsize=(20, 4))
      for i in range(n):
          ax = plt.subplot(1, n, i+1)
          plt.imshow(imgs_min[i].reshape(28, 28))
          plt.title('Model prediction: {} (p={:.2f}) \n Label: {} \n Trust score: {:.3f}' \
                    '\n Closest other class: {}'.format(pred_min[i].argmax(), pred_min[i].max(),
                                                         label_min[i], score_min[i], closest_
          ↪min[i]), fontsize=14)
          ax.get_xaxis().set_visible(False)
          ax.get_yaxis().set_visible(False)
      plt.show()
```



High Trust Scores

The high trust scores on the other hand all are very clear 1's:

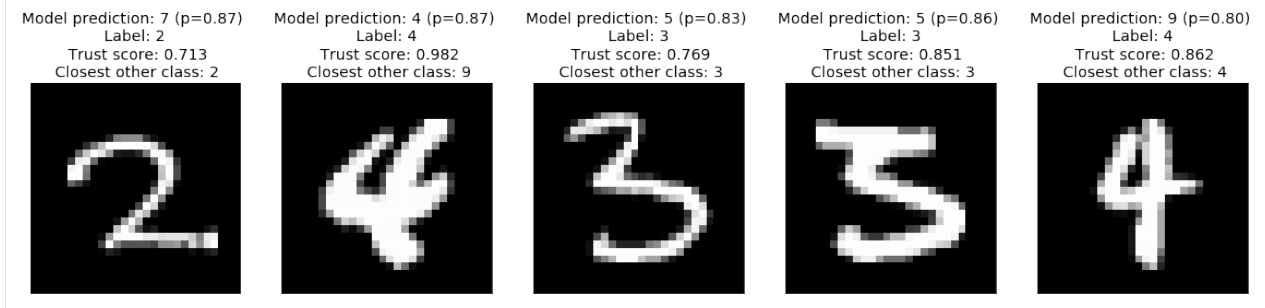
```
[17]: plt.figure(figsize=(20, 4))
      for i in range(n):
          ax = plt.subplot(1, n, i+1)
          plt.imshow(imgs_max[i].reshape(28, 28))
          plt.title('Model prediction: {} (p={:.2f}) \n Label: {} \n Trust score: {:.3f} \n \n Closest other class: {}'.format(pred_max[i].argmax(), pred_max[i].max(),
                                                                                                     label_max[i], score_max[i], closest_
←max[i]), fontsize=14)
          ax.get_xaxis().set_visible(False)
          ax.get_yaxis().set_visible(False)
      plt.show()
```

High model confidence, low trust score

Where trust scores really matter is when the predicted model confidence is relatively high (e.g. $p \in [0.8, 0.9]$) but the corresponding trust score is low, this can indicate samples for which the model is overconfident. The trust score provides a diagnostic for finding these examples:

```
[18]: plt.figure(figsize=(20, 4))
for i in range(min(n, len(idx_low))): # in case fewer than n instances are found
    ax = plt.subplot(1, n, i+1)
    plt.imshow(imgs_low[i].reshape(28, 28))
    plt.title('Model prediction: {} (p={:.2f}) \n Label: {} \n Trust score: {:.3f} \n \n Closest other class: {}'.format(pred_low[i].argmax(), pred_low[i].max(),
                                                                 label_low[i], score_low[i], closest_
    low[i]), fontsize=14)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



We can see several examples of an over-confident model predicting the wrong class, the low trust score, however, reveals that this is happening and the predictions should not be trusted despite the high model confidence.

In the following section we will see that on average trust scores outperform the model confidence for identifying correctly classified samples.

Comparison of Trust Scores with model prediction probabilities

Let's compare the prediction probabilities from the classifier with the trust scores for each prediction by checking whether trust scores are better than the model's prediction probabilities at identifying correctly classified examples.

First we need to set up a couple of helper functions.

- Define a function that handles model training and predictions:

```
[19]: def run_sc(X_train, y_train, X_test):
    clf = sc_model()
    clf.fit(X_train, y_train, batch_size=128, epochs=5, verbose=0)
    y_pred_proba = clf.predict(X_test)
    y_pred = np.argmax(y_pred_proba, axis=1)
    probas = y_pred_proba[range(len(y_pred)), y_pred] # probabilities of predicted class
    return y_pred, probas
```

- Define the function that generates the precision plots:

```
[20]: def plot_precision_curve(plot_title,
                               percentiles,
                               labels,
                               final_tp,
                               final_stderr,
                               final_misclassification,
                               colors = ['blue', 'darkorange', 'brown', 'red', 'purple']):

    plt.title(plot_title, fontsize=18)
    colors = colors + list(cm.rainbow(np.linspace(0, 1, len(final_tp))))
    plt.xlabel("Percentile", fontsize=14)
    plt.ylabel("Precision", fontsize=14)

    for i, label in enumerate(labels):
        ls = "--" if ("Model" in label) else "-"
        plt.plot(percentiles, final_tp[i], ls, c=colors[i], label=label)
        plt.fill_between(percentiles,
                         final_tp[i] - final_stderr[i],
                         final_tp[i] + final_stderr[i],
                         color=colors[i],
                         alpha=.1)

    if 0. in percentiles:
        plt.legend(loc="lower right", fontsize=14)
    else:
        plt.legend(loc="upper left", fontsize=14)
    model_acc = 100 * (1 - final_misclassification)
    plt.axvline(x=model_acc, linestyle="dotted", color="black")
    plt.show()
```

- The function below trains the model on a number of folds, makes predictions, calculates the trust scores, and generates the precision curves to compare the trust scores with the model prediction probabilities:

```
[21]: def run_precision_plt(X, y, nfolds, percentiles, run_model, test_size=.2,
                             plt_title="", plt_names=[], predict_correct=True, classes=10):
```

(continues on next page)

(continued from previous page)

```

def stderr(L):
    return np.std(L) / np.sqrt(len(L))

all_tp = [[[ for p in percentiles] for _ in plt_names]
misclassifications = []
mult = 1 if predict_correct else -1

folds = StratifiedShuffleSplit(n_splits=nfolds, test_size=test_size, random_state=0)
for train_idx, test_idx in folds.split(X, y):
    # create train and test folds, train model and make predictions
    X_train, y_train = X[train_idx, :], y[train_idx, :]
    X_test, y_test = X[test_idx, :], y[test_idx, :]
    y_pred, probas = run_model(X_train, y_train, X_test)
    # target points are the correctly classified points
    y_test_class = np.argmax(y_test, axis=1)
    target_points = (np.where(y_pred == y_test_class)[0] if predict_correct else
                    np.where(y_pred != y_test_class)[0])
    final_curves = [probas]
    # calculate trust scores
    ts = TrustScore()
    ts.fit(enc.predict(X_train), y_train, classes=classes)
    scores, _ = ts.score(enc.predict(X_test), y_pred, k=5)
    final_curves.append(scores) # contains prediction probabilities and trust scores
    # check where prediction probabilities and trust scores are above a certain
    ↪percentage level
    for p, perc in enumerate(percentiles):
        high_proba = [np.where(mult * curve >= np.percentile(mult * curve, perc))[0]]
    ↪for curve in final_curves]
        if 0 in map(len, high_proba):
            continue
        # calculate fraction of values above percentage level that are correctly (or
    ↪incorrectly) classified
        tp = [len(np.intersect1d(hp, target_points)) / (1. * len(hp)) for hp in high_
    ↪proba]
        for i in range(len(plt_names)):
            all_tp[i][p].append(tp[i]) # for each percentile, store fraction of
    ↪values above cutoff value
            misclassifications.append(len(target_points) / (1. * len(X_test)))

# average over folds for each percentile
final_tp = [[] for _ in plt_names]
final_stderr = [[] for _ in plt_names]
for p, perc in enumerate(percentiles):
    for i in range(len(plt_names)):
        final_tp[i].append(np.mean(all_tp[i][p]))
        final_stderr[i].append(stderr(all_tp[i][p]))

for i in range(len(all_tp)):
    final_tp[i] = np.array(final_tp[i])
    final_stderr[i] = np.array(final_stderr[i])

final_misclassification = np.mean(misclassifications)

```

(continues on next page)

(continued from previous page)

```
# create plot
plot_precision_curve(plt_title, percentiles, plt_names, final_tp, final_stderr,
    ↪final_misclassification)
```

Detect correctly classified examples

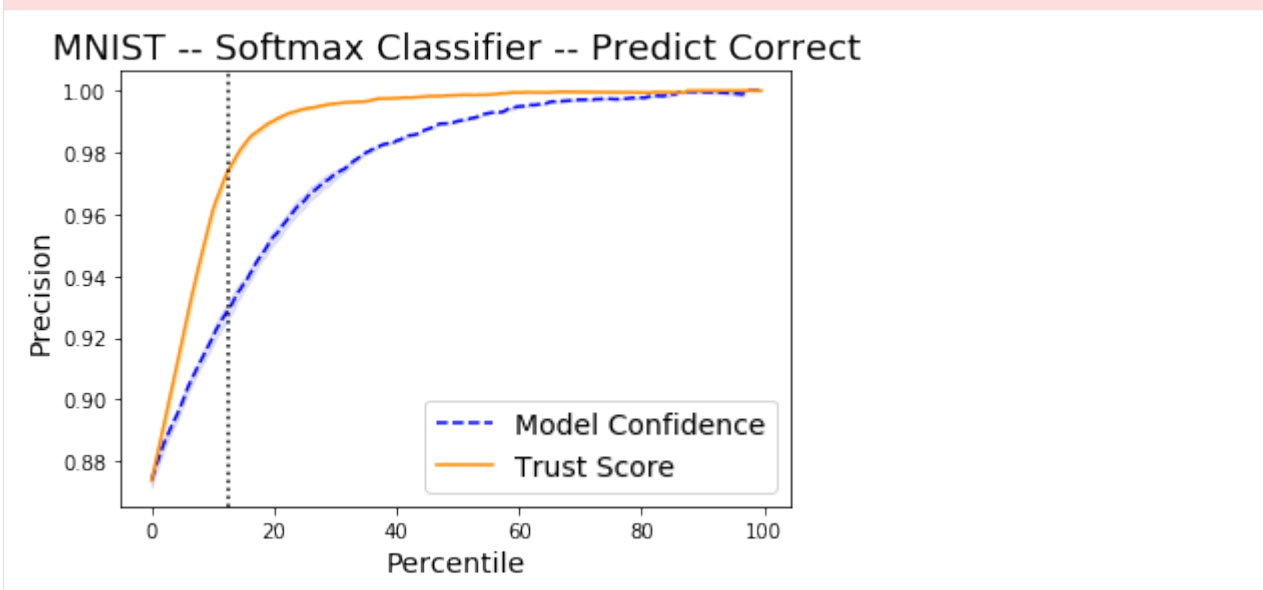
The x-axis on the plot below shows the percentiles for the model prediction probabilities of the predicted class for each instance and for the trust scores. The y-axis represents the precision for each percentile. For each percentile level, we take the test examples whose trust score is above that percentile level and plot the percentage of those points that were correctly classified by the classifier. We do the same with the classifier's own model confidence (i.e. softmax probabilities). For example, at percentile level 80, we take the top 20% scoring test examples based on the trust score and plot the percentage of those points that were correctly classified. We also plot the top 20% scoring test examples based on model probabilities and plot the percentage of those that were correctly classified. The vertical dotted line is the error of the classifier. The plots are an average over 2 folds of the dataset with 20% of the data kept for the test set.

The *Trust Score* and *Model Confidence* curves then show that the model precision is typically higher when using the trust scores to rank the predictions compared to the model prediction probabilities.

```
[22]: X = x_train
      y = y_train
      percentiles = [0 + 0.5 * i for i in range(200)]
      nfolds = 2
      plt_names = ['Model Confidence', 'Trust Score']
      plt_title = 'MNIST -- Softmax Classifier -- Predict Correct'
```

```
[23]: run_precision_plt(X, y, nfolds, percentiles, run_sc, plt_title=plt_title,
      plt_names=plt_names, predict_correct=True)
```

Reshaping data from (48000, 4, 4, 4) to (48000, 64) so k-d trees can be built.
 Reshaping data from (12000, 4, 4, 4) to (12000, 64) so k-d trees can be queried.
 Reshaping data from (48000, 4, 4, 4) to (48000, 64) so k-d trees can be built.
 Reshaping data from (12000, 4, 4, 4) to (12000, 64) so k-d trees can be queried.



[source]

11.1 ProtoSelect

11.1.1 Overview

Bien and Tibshirani (2012) proposed ProtoSelect, which is a prototype selection method with the goal of constructing not only a condensed view of a dataset but also an interpretable model (applicable to classification only). Prototypes can be defined as instances that are representative of the entire training data distribution. Formally, consider a dataset of training points $\mathcal{X} = \{x_1, \dots, x_n\} \subset \mathbf{R}^p$ and their corresponding labels $\mathcal{Y} = \{y_1, \dots, y_n\}$, where $y_i \in \{1, 2, \dots, L\}$. ProtoSelect finds sets $\mathcal{P}_l \subseteq \mathcal{X}$ for each class l such that the set union of $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_L$ would provided a distilled view of the training dataset $(\mathcal{X}, \mathcal{Y})$.

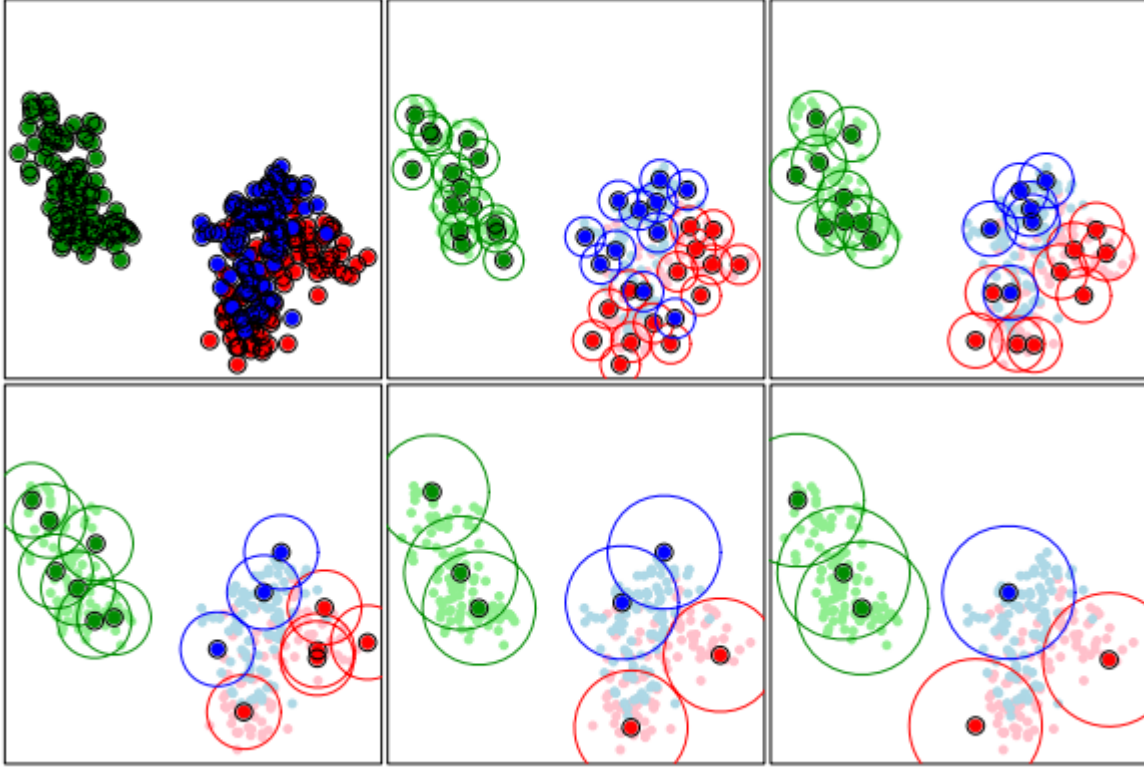
Given the sets of prototypes, one can construct a simple interpretable classifier given by:

$$\hat{c}(x) = \operatorname{argmin}_l \min_{z \in \mathcal{P}_l} d(x, z)$$

Note that the classifier defined in the equation above would be equivalent to 1-KNN if each set \mathcal{P}_l would consist only of instances belonging to class l .

11.1.2 ProtoSelect method

ProtoSelect is designed such that each prototype would satisfy a set of desired properties. For a set $\mathcal{P}_l \subseteq \mathcal{X}$, the neighborhood of a point $x_i \in \mathcal{P}_l$ is given by the points contained in an ϵ -ball centered in x_i , denoted as $B(x_i, \epsilon)$. Thus, given a radius ϵ for a point x_i , we say that another point x_j is covered by x_i if x_j is contained in the ϵ -ball centered on x_i . A visualization of the prototypes sets for various ϵ radius values are depicted in the following figure:



Bien and Tibshirani, *PROTOTYPE SELECTION FOR INTERPRETABLE CLASSIFICATION*, 2012

A desirable prototype set for a class l would satisfy the following properties:

- cover as many training points as possible of the class l .
- covers as few training points as possible of classes different from l .
- is sparse (i.e., contains as few prototypes instances as possible).

Formally, let us first define $\alpha_j^{(l)} \in \{0, 1\}$ to indicate whether we select x_j to be in \mathcal{P}_l . Then we can write the three properties as an [integer program](#) as follows:

$$\min_{\alpha_j^{(l)}, \xi_i, \nu_i} \sum_i \xi_i + \sum_i \nu_i + \lambda \sum_{j,l} \alpha_j^{(l)} \text{ such that} \quad (11.1)$$

$$\sum_{j: x_i \in B(x_j, \epsilon)} \alpha_j^{(y_i)} \geq 1 - \xi_i, \forall x_i \in \mathcal{X}, \quad (a)$$

$$\sum_{j: x_i \in B(x_j, \epsilon), l \neq y_i} \alpha_j^{(l)} \leq 0 + \nu_i, \forall x_i \in \mathcal{X}, \quad (b)$$

$$\alpha_j^l \in \{0, 1\} \forall j, l, \quad (11.2)$$

$$\xi_i, \nu_i \geq 0. \quad (11.3)$$

For each training point x_i , we introduce the slack variables ξ_i and ν_i . Before explaining the two constraints, note that $\sum_{j: x_i \in B(x_j, \epsilon)} \alpha_j^{(l)}$ counts the number of balls $B(x_j, \epsilon)$ with $x_j \in \mathcal{P}_l$ that cover the point x_i . The constraint (a) tries

to encourage that for each training point (x_i, y_i) , x_i is covered in at least one ϵ -ball of a prototype for the class y_i . On the other hand, the constraint (b) tries to encourage that x_i will not belong to any ϵ -ball centered in a prototype for the other classes $l \neq y_i$.

Because the integer program defined above cannot be solved in polynomial time, the authors propose two alternative solution. The first one consists of a relaxation of the objective and a transformation of the integer program into a linear program, for which post-processing is required to ensure feasibility of the solution. We refer the reader to the [paper](#) for more details. The second one, recommended and implemented in *Alibi*, follows a greedy approach. Given the current choice of prototypes subsets $(\mathcal{P}_1, \dots, \mathcal{P}_L)$, in the next iteration we update it to $(\mathcal{P}_1, \dots, \mathcal{P}_l \cup \{x_j\}, \dots, \mathcal{P}_L)$, where x_j is selected such that it maximizes the objective $\Delta Obj(x_j, l) = \Delta\xi(x_j, l) - \Delta\nu(x_j, l) - \lambda$, where:

$$\Delta\xi(x_j, l) = |\mathcal{X}_l \cap (B(x_j, \epsilon) \setminus \cup_{x_{j'} \in \mathcal{P}_l} B(x_{j'}, \epsilon))| \quad (a)$$

$$\Delta\nu(x_j, l) = |B(x_j, \epsilon) \cap (\mathcal{X} \setminus \mathcal{X}_l)|. \quad (b)$$

Note that $\Delta\xi(x_j, l)$ counts the number of new instances (i.e. not already covered by the existing prototypes) belonging to class l that x_j covers in the ϵ -ball. On the other hand, $\Delta\nu(x_j, l)$ counts how many instances belonging to a different class than l the x_j element covers. Finally, λ is the penalty/cost of adding a new prototypes encouraging sparsity (lower number of prototypes). Intuitively, a good prototype for a class l will cover as many new instances belonging to class l (i.e. maximize $\Delta\xi(x_j, l)$) and avoid covering elements outside the class l (i.e. minimize $\Delta\nu(x_j, l)$). The prototype selection algorithm stops when all $\Delta Obj(x_j, l)$ are lower than 0.

11.1.3 Usage

```
from alibi.prototypes import ProtoSelect
from alibi.utils.kernel import EuclideanDistance

summariser = ProtoSelect(kernel_distance=EuclideanDistance(), eps=eps, preprocess_
↪ fn=preprocess_fn)
```

- **kernel_distance**: Kernel distance to be used. Expected to support computation in batches. Given an input x of size $N_x \times f_1 \times f_2 \times \dots$ and an input y of size $N_y \times f_1 \times f_2 \times \dots$, the kernel distance should return a kernel matrix of size $N_x \times N_y$.
- **eps**: Epsilon ball size.
- **lambda_penalty**: Penalty for each prototype. Encourages a lower number of prototypes to be selected. Corresponds to λ in the paper's notation. If not specified, the default value is set to $1 / N$, where N is the size of the dataset to choose the prototype instances from, passed to the [fit](#) method.
- **batch_size**: Batch size to be used for kernel matrix computation.
- **preprocess_fn**: Preprocessing function used for kernel matrix computation. The preprocessing function takes the input in a `list` or a `numpy` array and transforms it into a `numpy` array which is then fed to the `kernel_distance` function. The use of `preprocess_fn` allows the method to be applied to any data modality.
- **verbose**: Whether to display progression bar while computing prototype points.

Following the initialization, we need to fit the `summariser`.

```
summariser = summariser.fit(X=X_train, y=y_train)
```

- **X**: Dataset to be summarised.
- **y**: Labels of the dataset X. The labels are expected to be represented as integers `[0, 1, ..., L-1]`, where L is the number of classes in the dataset X.

In a more general case, we can specify an optional dataset Z to choose the prototypes from (see the documentation of the `fit` method). In this scenario, the dataset to be summarised is still X , but it is summarised by prototypes belonging to the dataset Z . Furthermore, note that we only need to specify the labels for the X set through y , but not for Z . In case the labels y are missing, the method implicitly assumes that all the instances belong to the same class. This means that the second term in the objective, $\Delta\nu(x_j, l)$, will be 0. Thus, the algorithm will try to find prototypes that cover as many data instances as possible, with minimum overlap between their corresponding ϵ -balls.

Finally, we can obtain a summary by requesting the maximum number of prototypes to be returned:

```
summary = summariser.summarise(num_prototypes=num_prototypes)
```

- `num_prototypes`: Maximum number of prototypes to be selected.

As we previously mentioned, the algorithm stops when the objective is less than 0, for all the remaining instances in the set of potential prototypes. This means that the algorithm can return a lower number of prototypes than the one requested.

Another important observation is that the summary returns the prototypes with their corresponding labels although no labels were provided for Z . This is possible since each prototype z will belong to a prototype set \mathcal{P}_l , and thus we can assign a label l to z . Following the summarisation step, one can train an interpretable 1-KNN classifier on the returned prototypes even for an unlabeled dataset Z .

Warning

If the optional argument Z is not provided, it is automatically set to X . Although the labels of the data instances belonging to Z are available in this case, the dataset Z is still viewed as an unlabeled dataset. This means that a prototype $z_i \in Z$ belonging to the class l according to the labels y , can be a prototype for a class $k \neq l$.

11.1.4 Hyperparameter selection

Alibi exposes a cross-validation hyperparameter selection method for the radius ϵ when the Euclidean distance is used. The method returns the ϵ radius value that achieves the best accuracy score on a 1-KNN classification task.

```
from alibi.prototypes.protoselect import cv_protoselect_euclidean

cv = cv_protoselect_euclidean(trainset=(X_train, y_train),
                              valset=(X_val, y_val),
                              num_prototypes=num_prototypes,
                              quantiles=(0., 0.4),
                              preprocess_fn=preprocess_fn)
```

The method API is flexible and allows for various arguments to be passed such as a predefined ϵ -grid, the number of equidistant bins, keyword arguments to the KFold split when the validation set is not provided, etc. We refer the reader to the [documentation page](#) for a full parameter description.

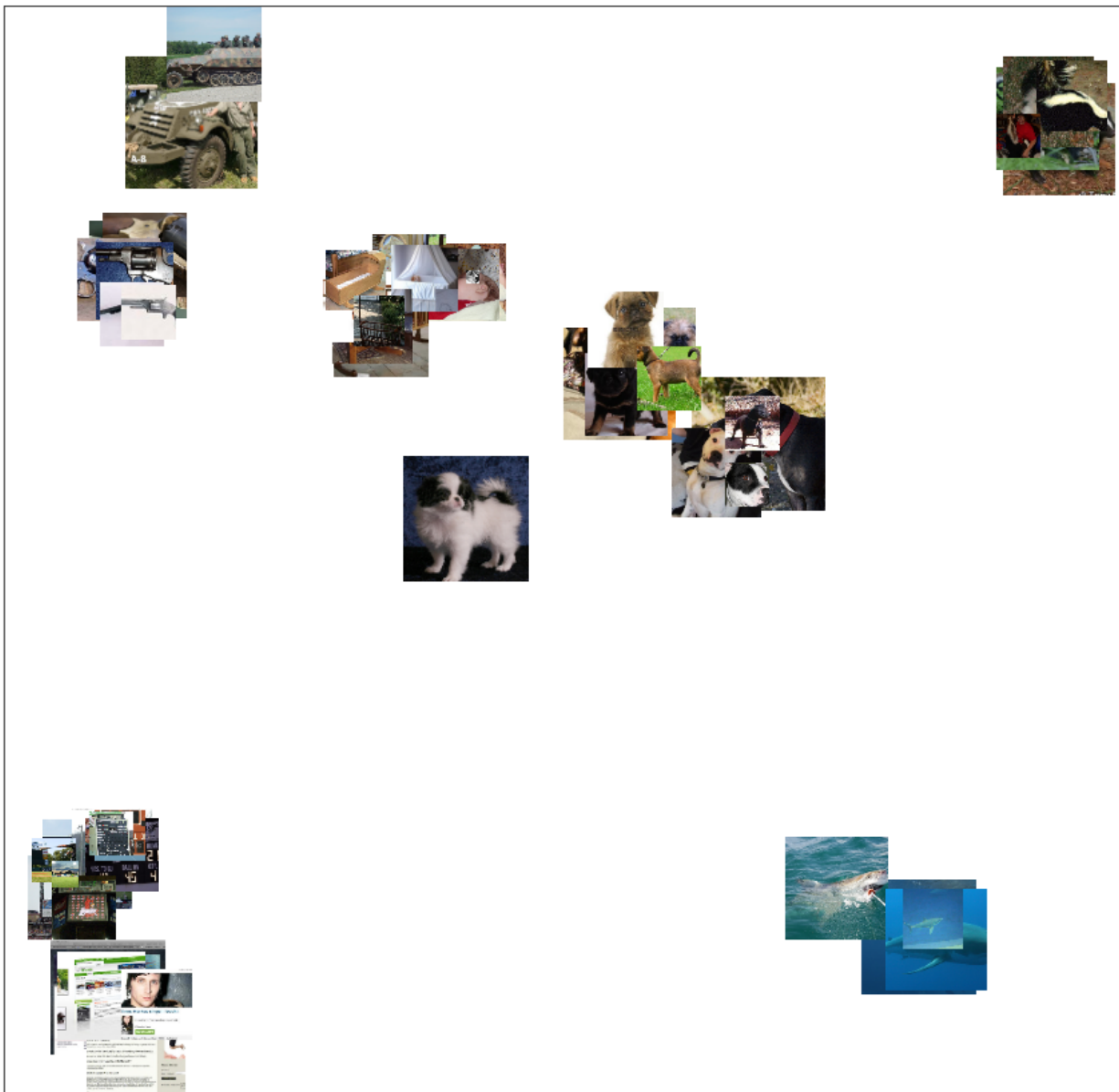
The best ϵ -radius can be access through `cv['best_eps']`. The object also contains other metadata gathered throughout the hyperparameter search.

11.1.5 Data modalities

The method can be applied to any data modality by passing the `preprocess_fn: Callable[[Union[list, np.ndarray]], np.ndarray]` expected to return a numpy array feature representation compatible with the kernel provided.

11.1.6 Prototypes visualization for image modality

As proposed by [Bien and Tibshirani \(2012\)](#), one can visualize and understand the importance of a prototype in a 2D image scatter plot. To obtain the image size of each prototype, we fit a 1-KNN classifier on the prototypes using the feature representation provided by `preprocess_fn` and the Euclidean distance metric, which is consistent with our choice of kernel dissimilarity. The size of each prototype is proportional to the logarithm of the number of assigned training instances correctly classified according to the 1-KNN classifier. Thus, the larger the image, the more important the prototype is.



Prototypes of a subsampled ImageNet dataset containing 10 classes using a ResNet50 pretrained feature extractor.

```
import umap
from alibi.prototypes import visualize_prototypes

# define 2D reducer
reducer = umap.UMAP(random_state=26)
reducer = reducer.fit(preprocess_fn(X_train))

# display prototypes in 2D
visualize_image_prototypes(summary=summary,
                           trainset=(X_train, y_train),
                           reducer=reducer.transform,
                           preprocess_fn=preprocess_fn)
```

- `summary`: An `Explanation` object produced by a call to the `summarise` method.
- `trainset`: Tuple, `(X_train, y_train)`, consisting of the training data instances with the corresponding labels.
- `reducer`: 2D reducer. Reduces the input feature representation to 2D. Note that the reducer operates directly on the input instances if `preprocess_fn=None`. If the `preprocess_fn` is specified, the reducer will be called on the feature representation obtained after calling `preprocess_fn` on the input instances.
- `preprocess_fn`: Preprocessor function.

Here we used a `UMAP` 2D reducer, but any other dimensionality reduction method will do. The `visualize_image_prototypes` method exposes other arguments to control how the images will be displayed. We refer the reader to the [documentation page](#) for further details.

11.1.7 Examples

Tabular and image datasets

EXAMPLES

12.1 ProtoSelect

12.1.1 ProtoSelect on Adult Census and CIFAR10

Bien and Tibshirani (2012) proposed ProtoSelect, which is a prototype selection method with the goal of constructing not only a condensed view of a dataset but also an interpretable model (applicable to classification only). Prototypes can be defined as instances that are representative of the entire training data distribution. Formally, consider a dataset of training points $\mathcal{X} = \{x_1, \dots, x_n\} \subset \mathbf{R}^p$ and their corresponding labels $\mathcal{Y} = \{y_1, \dots, y_n\}$, where $y_i \in \{1, 2, \dots, L\}$. ProtoSelect finds sets $\mathcal{P}_l \subseteq \mathcal{X}$ for each class l such that the set union of $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_L$ would provide a distilled view of the training dataset $(\mathcal{X}, \mathcal{Y})$.

Given the sets of prototypes, one can construct a simple interpretable classifier given by:

$$\hat{c}(x) = \operatorname{argmin}_l \min_{z \in \mathcal{P}_l} d(x, z)$$

Note that the classifier defined in the equation above would be equivalent to 1-KNN if each set \mathcal{P}_l would consist only of instances belonging to class l .

```
[1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from typing import List, Dict, Tuple

import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.layers as layers

from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.neighbors import KNeighborsClassifier

from alibi.api.interfaces import Explanation
from alibi.datasets import fetch_adult
from alibi.prototypes import ProtoSelect, visualize_image_prototypes
from alibi.utils.kernel import EuclideanDistance
from alibi.prototypes.protoselect import cv_protoselect_euclidean
```

Utils

Utility function to display the tabular data in a human-readable format.

```
[2]: def display_table(X: np.ndarray,
                      y: np.ndarray,
                      feature_names: List[str],
                      category_map: Dict[int, List[str]],
                      target_names: List[str]) -> pd.DataFrame:
    """
    Displays the table in a human readable format.

    Parameters
    -----
    X
        Array of data instances to be displayed.
    y
        Array of data labels.
    feature_names
        List of feature names.
    category_map
        Category mapping dictionary having as keys the categorical index and as values
    ↪ the categorical values
        each feature takes.
    target_names
        List of label names.

    Return
    -----
    `DataFrame` containing the concatenation of `X` and `Y` in a human readable format.
    """
    # concat labels to the original instances
    orig = np.concatenate([X, y.reshape(-1, 1)], axis=1)

    # define new feature names and category map by including the label
    feature_names = feature_names + ["Label"]
    category_map.update({feature_names.index("Label"): [target_names[i] for i in np.
    ↪ unique(y)]})

    # replace label encodings with strings
    df = pd.DataFrame(orig, columns=feature_names)
    for key in category_map:
        df[feature_names[key]].replace(range(len(category_map[key])), category_map[key],
    ↪ inplace=True)

    dfs = []
    for l in np.unique(y):
        dfs.append(df[df['Label'] == target_names[l]])

    return pd.concat(dfs)
```

Utility function to display image prototypes.

```
[3]: def display_images(summary: Explanation, imgsize: float = 1.5):
    """
    Displays image prototypes per class.

    Parameters
    -----
    summary
        An `Explanation` object produced by a call to the `summarise` method.
    imgsize
        Image size of a prototype.
    """
    X_protos, y_protos = summary.data['prototypes'], summary.data['prototype_labels']
    str_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
    ↪ 'ship', 'truck']
    int_labels, counts = np.unique(y_protos, return_counts=True)
    max_counts = np.max(counts).item()

    fig, axs = plt.subplots(len(int_labels), max_counts)
    fig.set_figheight(len(int_labels) * imgsize)
    fig.set_figwidth(max_counts * imgsize)

    for i, l in enumerate(int_labels):
        indices = np.where(y_protos == l)[0]
        X = X_protos[indices]

        for j in range(max_counts):
            if j < len(indices):
                axs[i][j].imshow(X[j])
            else:
                fig.delaxes(axs[i][j])

        axs[i][j].set_xticks([])
        axs[i][j].set_yticks([])

        axs[i][0].set_ylabel(str_labels[l])
```

Adult Census dataset

Load Adult Census dataset

Fetch the *Adult Census* dataset and perform train-test-validation split. In this example, for demonstrative purposes, each split contains only 1000. One can increase the number of instances in each set but should be aware of the memory limitation since the kernel matrix used for prototype selection is precomputed and stored in memory.

```
[4]: # fetch adult datasets
adult = fetch_adult()

# split dataset into train-test-validation
X, y = adult.data, adult.target
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=1000, test_
↪ size=2000, random_state=13)
```

(continues on next page)

(continued from previous page)

```
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size=0.5, random_
↪state=13)

# identify numerical and categorical columns
categorical_ids = list(adult.category_map.keys())
numerical_ids = [i for i in range(len(adult.feature_names)) if i not in adult.category_
↪map]
```

Preprocessing function

Because the tabular dataset has low dimensionality, we can use a simple preprocessing function: numerical features are standardized and categorical features are one-hot encoded. The kernel dissimilarity used for prototype selection will operate on the preprocessed representation.

```
[5]: # define data preprocessor
num_transf = StandardScaler()
cat_transf = OneHotEncoder(
    categories=[range(len(x)) for x in adult.category_map.values()],
    handle_unknown='ignore'
)
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', cat_transf, categorical_ids),
        ('num', num_transf, numerical_ids)
    ],
    sparse_threshold=0
)

# fit data preprocessor
preprocessor = preprocessor.fit(adult.data)
```

Prototypes selection

As with every kernel-based method, the performance of **ProtoSelect** is sensitive to the kernel selection and a predefined ϵ -radius which characterizes the neighborhood of an instance x as a hyper-sphere of radius ϵ centered in x denoted as $B(x_i, \epsilon)$. Note that other kernel dissimilarities might require some tuning (e.g., Gaussian RBF), which means that we will have to jointly search for the optimum ϵ and kernel parameters. Luckily, in our case, we will use a simple Euclidean distance metric that does not require any tuning. Thus, we only need to search for the optimum ϵ -radius to be used by **ProtoSelect**. *Alibi* already comes with support for a grid-based search of the optimum values of the ϵ when using a Euclidean distance metric.

To search for the optimum ϵ -radius, we call the `cv_protoselect_euclidean` method, provided with a training dataset, an optional prototype dataset (i.e. training dataset is used by default if prototype dataset is not provided), and a validation set. Note that in the absence of a validation dataset, the method performs cross-validation on the training dataset.

```
[6]: num_prototypes = 20
grid_size = 50
quantiles = (0., .5)

# search for the best epsilon-radius value
```

(continues on next page)

(continued from previous page)

```
cv = cv_protoselect_euclidean(trainset=(X_train, y_train),
                             valset=(X_val, y_val),
                             num_prototypes=num_prototypes,
                             quantiles=quantiles,
                             grid_size=grid_size,
                             preprocess_fn=preprocessor.transform)
```

Once we have the optimum value of ϵ , we can instantiate **ProtoSelect** as follows:

```
[7]: summariser = ProtoSelect(kernel_distance=EuclideanDistance(),
                             eps=cv['best_eps'],
                             preprocess_fn=preprocessor.transform)

summariser = summariser.fit(X=X_train, y=y_train)
summary = summariser.summarise(num_prototypes=num_prototypes)
print(f"Found {len(summary.data['prototypes'])} prototypes.")

Found 20 prototypes.
```

Display prototypes

Let us inspect the returned prototypes:

```
[8]: X_protos = summary.data['prototypes']
y_protos = summary.data['prototype_labels']

# display the prototypes in a human readable format
display_table(X=X_protos,
              y=y_protos,
              feature_names=adult.feature_names,
              category_map=adult.category_map,
              target_names=adult.target_names)
```

```
[8]:
```

	Age	Workclass	Education	Marital Status	Occupation \
0	33	Private	High School grad	Never-Married	Blue-Collar
1	31	Private	High School grad	Never-Married	Service
2	60	Private	Associates	Separated	Service
3	61	?	High School grad	Married	?
4	20	?	High School grad	Never-Married	?
5	31	Private	High School grad	Never-Married	Service
6	27	?	High School grad	Separated	?
7	63	Private	Dropout	Widowed	Service
8	67	Federal-gov	Bachelors	Widowed	Admin
9	25	Private	Dropout	Never-Married	Service
10	38	Self-emp-not-inc	Bachelors	Never-Married	Blue-Collar
11	31	Private	High School grad	Separated	Blue-Collar
12	21	Private	High School grad	Never-Married	Sales
13	17	?	Dropout	Never-Married	?
14	61	Local-gov	Masters	Married	White-Collar
15	51	Self-emp-inc	Doctorate	Married	Professional
16	55	Private	Masters	Married	White-Collar
17	33	Private	Bachelors	Married	Professional

(continues on next page)

(continued from previous page)

18	49	Self-emp-inc	Prof-School	Married	Professional	
19	90	Private	Prof-School	Married	Professional	
		Relationship	Race	Sex	Capital Gain	Capital Loss \
0		Not-in-family	White	Male	0	0
1		Own-child	White	Female	0	0
2		Not-in-family	White	Female	0	0
3		Husband	White	Male	0	0
4		Own-child	White	Male	0	0
5		Own-child	White	Male	0	1721
6		Own-child	Black	Female	0	0
7		Not-in-family	White	Female	0	0
8		Not-in-family	White	Female	0	0
9		Unmarried	Black	Female	0	0
10		Not-in-family	Amer-Indian-Eskimo	Male	0	0
11		Unmarried	White	Female	0	2238
12		Own-child	Asian-Pac-Islander	Male	0	0
13		Own-child	White	Male	0	0
14		Husband	White	Male	7298	0
15		Husband	White	Male	15024	0
16		Husband	White	Male	0	1977
17		Husband	White	Male	15024	0
18		Husband	White	Male	99999	0
19		Husband	White	Male	20051	0
		Hours per week	Country	Label		
0		40	United-States	<=50K		
1		30	United-States	<=50K		
2		40	United-States	<=50K		
3		6	United-States	<=50K		
4		20	United-States	<=50K		
5		16	United-States	<=50K		
6		40	United-States	<=50K		
7		31	United-States	<=50K		
8		40	United-States	<=50K		
9		32	United-States	<=50K		
10		30	United-States	<=50K		
11		40	United-States	<=50K		
12		30	British-Commonwealth	<=50K		
13		20	South-America	<=50K		
14		60	United-States	>50K		
15		40	United-States	>50K		
16		40	United-States	>50K		
17		75	United-States	>50K		
18		37	United-States	>50K		
19		72	United-States	>50K		

By inspecting the prototypes, we can observe that features like Education and Marital Status can reveal some patterns. People with lower education level (e.g., High School grad, Dropout, etc.) and who don't have a partner (e.g., Never-Married, Separated, Widowed etc.) tend to be classified as $\leq 50K$. On the other hand, we have people that have a higher education level (e.g., Bachelors, Masters, Doctorate, etc.) and who have a partner (e.g., Married) that are classified as $> 50K$.

Train 1-KNN

A standard procedure to check the quality of the prototypes is to train a 1-KNN classifier and evaluate its performance.

```
[9]: # train 1-knn classifier using the selected prototypes
knn_proto = KNeighborsClassifier(n_neighbors=1, metric='euclidean')
knn_proto = knn_proto.fit(X=preprocessor.transform(X_protos), y=y_protos)
```

To verify that **ProtoSelect** returns better prototypes than a simple random selection, we randomly sample multiple prototype sets, train a 1-KNN for each set, evaluate the classifiers, and return the average accuracy score.

```
[10]: np.random.seed(0)
scores = []

for i in range(10):
    rand_idx = np.random.choice(len(X_train), size=len(X_protos), replace=False)
    rands, rands_labels = X_train[rand_idx], y_train[rand_idx]

    knn_rand = KNeighborsClassifier(n_neighbors=1, metric='euclidean')
    knn_rand = knn_rand.fit(X=preprocessor.transform(rands), y=rands_labels)
    scores.append(knn_rand.score(preprocessor.transform(X_test), y_test))
```

Compare the results returned by **ProtoSelect** and by the random sampling.

```
[11]: # compare the scores obtained by ProtoSelect vs random choice
print('ProtoSelect 1-KNN accuracy: %.3f' % (knn_proto.score(preprocessor.transform(X_
↪test), y_test)))
print('Random 1-KNN mean accuracy: %.3f' % (np.mean(scores)))

ProtoSelect 1-KNN accuracy: 0.813
Random 1-KNN mean accuracy: 0.723
```

We can observe that **ProtoSelect** chooses more representative instances than a naive random selection. The gap between the two should narrow as we increase the number of requested prototypes.

CIFAR10 dataset

Load dataset

Fetch the *CIFAR10* dataset and perform train-test-validation split and standard preprocessing. For demonstrative purposes, we use a reduced dataset and remind the user about the memory limitation of pre-computing and storing the kernel matrix in memory.

```
[12]: (X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype(np.float32) / 255.
X_test = X_test.astype(np.float32) / 255.
y_train, y_test = y_train.flatten(), y_test.flatten()

# split the test into test-validation
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, train_size=1000, test_
↪size=1000, random_state=13)
```

(continues on next page)

(continued from previous page)

```
# subsample the datasets
np.random.seed(13)
train_idx = np.random.choice(len(X_train), size=1000, replace=False)
X_train, y_train = X_train[train_idx], y_train[train_idx]
```

Preprocessing function

For *CIFAR10*, we use a hidden layer output from a pre-trained network as our feature representation of the input images. The network was trained on the CIFAR10 dataset. Note that one can use any feature representation of choice (e.g., used from some self-supervised task).

```
[13]: # download weights
!wget https://storage.googleapis.com/seldon-models/alibi/model_cifar10/checkpoint -P
↪model_cifar10
!wget https://storage.googleapis.com/seldon-models/alibi/model_cifar10/cifar10.ckpt.data-
↪000000-of-000001 -P model_cifar10
!wget https://storage.googleapis.com/seldon-models/alibi/model_cifar10/cifar10.ckpt.
↪index -P model_cifar10

--2022-05-09 14:21:49-- https://storage.googleapis.com/seldon-models/alibi/model_
↪cifar10/checkpoint
Resolving storage.googleapis.com (storage.googleapis.com)... 142.250.178.16, 142.250.187.
↪240, 142.250.187.208, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.250.178.16|:443...
↪connected.
HTTP request sent, awaiting response... 200 OK
Length: 81 [application/octet-stream]
Saving to: 'model_cifar10/checkpoint'

checkpoint          100%[=====>]          81  --.-KB/s    in 0s

2022-05-09 14:21:49 (11,2 MB/s) - 'model_cifar10/checkpoint' saved [81/81]

--2022-05-09 14:21:49-- https://storage.googleapis.com/seldon-models/alibi/model_
↪cifar10/cifar10.ckpt.data-000000-of-000001
Resolving storage.googleapis.com (storage.googleapis.com)... 142.250.178.16, 142.250.187.
↪240, 142.250.187.208, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.250.178.16|:443...
↪connected.
HTTP request sent, awaiting response... 200 OK
Length: 2218891 (2,1M) [application/octet-stream]
Saving to: 'model_cifar10/cifar10.ckpt.data-000000-of-000001'

cifar10.ckpt.data-0 100%[=====>]    2,12M  --.-KB/s    in 0,08s

2022-05-09 14:21:49 (25,5 MB/s) - 'model_cifar10/cifar10.ckpt.data-000000-of-000001' saved
↪[2218891/2218891]

--2022-05-09 14:21:50-- https://storage.googleapis.com/seldon-models/alibi/model_
↪cifar10/cifar10.ckpt.index
Resolving storage.googleapis.com (storage.googleapis.com)... 142.250.178.16, 142.250.187.
```

(continues on next page)

(continued from previous page)

```

↪240, 142.250.187.208, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.250.178.16|:443...
↪connected.
HTTP request sent, awaiting response... 200 OK
Length: 2828 (2,8K) [application/octet-stream]
Saving to: 'model_cifar10/cifar10.ckpt.index'

cifar10.ckpt.index  100%[=====>]    2,76K  --.-KB/s    in 0s

2022-05-09 14:21:50 (38,4 MB/s) - 'model_cifar10/cifar10.ckpt.index' saved [2828/2828]

```

```

[14]: # define the network to be used.
model = keras.Sequential([
    layers.InputLayer(input_shape=(32, 32, 3)),
    layers.Conv2D(32, (3, 3), kernel_initializer='he_uniform', padding='same'),
    layers.ReLU(),
    layers.BatchNormalization(),
    layers.Conv2D(32, (3, 3), kernel_initializer='he_uniform', padding='same'),
    layers.ReLU(),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.2),
    layers.Conv2D(64, (3, 3), kernel_initializer='he_uniform', padding='same'),
    layers.ReLU(),
    layers.BatchNormalization(),
    layers.Conv2D(64, (3, 3), kernel_initializer='he_uniform', padding='same'),
    layers.ReLU(),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),
    layers.Conv2D(128, (3, 3), kernel_initializer='he_uniform', padding='same'),
    layers.ReLU(),
    layers.BatchNormalization(),
    layers.Conv2D(128, (3, 3), kernel_initializer='he_uniform', padding='same'),
    layers.ReLU(),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.4),
    layers.Flatten(),
    layers.Dense(128, kernel_initializer='he_uniform', name='feature_layer'),
    layers.ReLU(),
    layers.BatchNormalization(),
    layers.Dropout(0.5),
    layers.Dense(10)
])

# load the weights
model.load_weights('model_cifar10/cifar10.ckpt');

```

```
[15]: # define preprocessing function
partial_model = keras.Model(
    inputs=model.inputs,
    outputs=model.get_layer(name='feature_layer').output
)

def preprocess_fn(x: np.ndarray):
    return partial_model(x, training=False).numpy()
```

Prototypes selection

To obtain the best results, we apply the same grid-search procedure as in the case of the tabular dataset.

```
[16]: num_prototypes = 50
grid_size = 50
quantiles = (0.0, 0.5)

# get best eps by cv
cv = cv_protoselect_euclidean(trainset=(X_train, y_train),
                              valset=(X_val, y_val),
                              num_prototypes=num_prototypes,
                              quantiles=quantiles,
                              grid_size=grid_size,
                              preprocess_fn=preprocess_fn,
                              batch_size=100)
```

Once we have the optimum value of ϵ we can instantiate **ProtoSelect** as follows:

```
[17]: summariser = ProtoSelect(kernel_distance=EuclideanDistance(),
                               eps=cv['best_eps'],
                               preprocess_fn=preprocess_fn)

summariser = summariser.fit(X=X_train, y=y_train)
summary = summariser.summarise(num_prototypes=num_prototypes)
print(f"Found {len(summary.data['prototypes'])} prototypes.")

Found 31 prototypes.
```

Display prototypes

We can visualize and understand the importance of a prototype in a 2D image scatter plot. *Alibi* provides a helper function which fits a 1-KNN classifier on the prototypes and computes their importance as the logarithm of the number of assigned training instances correctly classified according to the 1-KNN classifier. Thus, the larger the image, the more important the prototype is.

```
[ ]: !pip install umap-learn
```

```
[18]: import umap

# define 2D reducer
```

(continues on next page)

(continued from previous page)

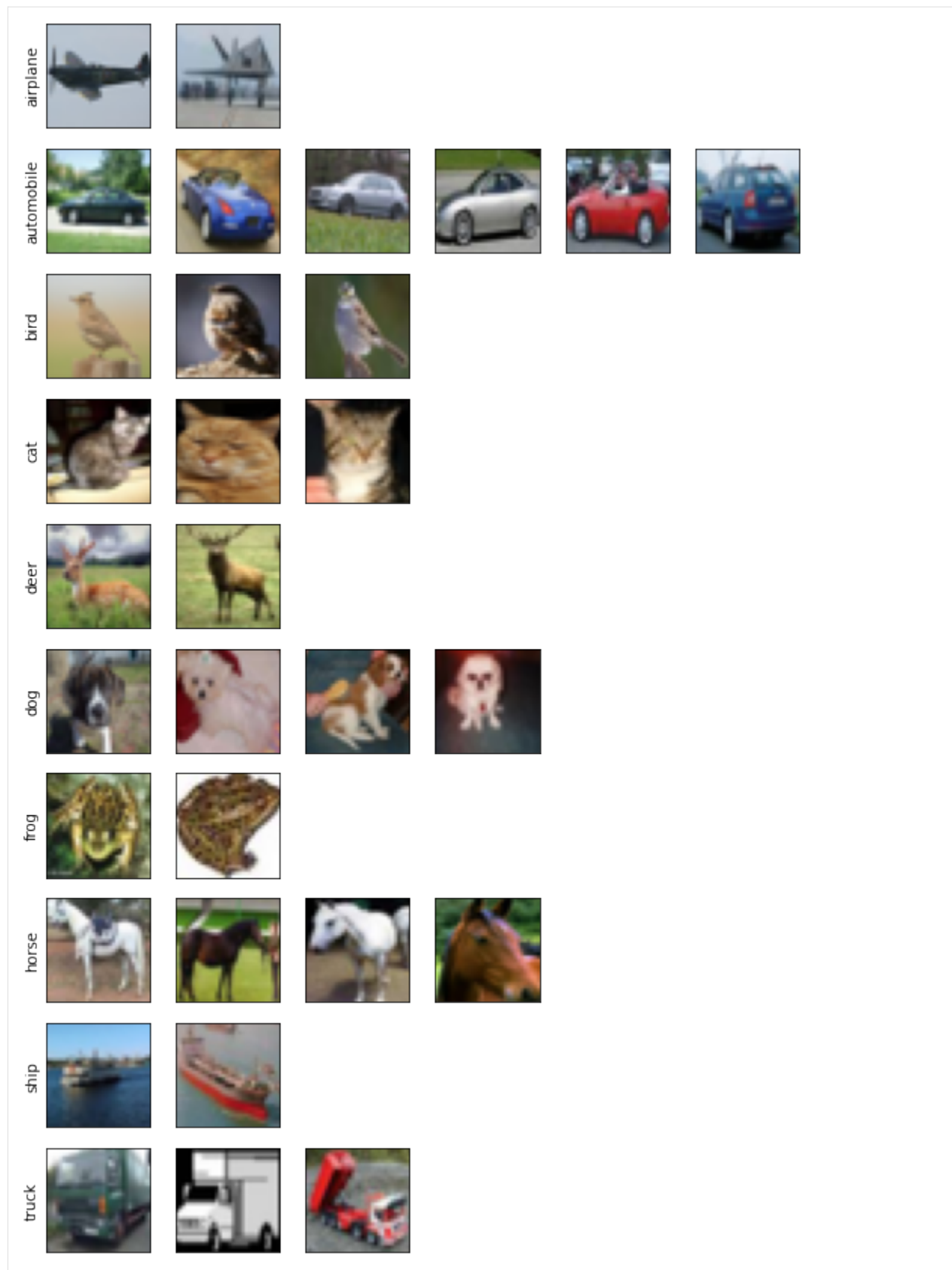
```
reducer = umap.UMAP(random_state=26)
reducer = reducer.fit(preprocess_fn(X_train))
```

```
[19]: # display prototypes in 2D
ax = visualize_image_prototypes(summary=summary,
                                trainset=(X_train, y_train),
                                reducer=reducer.transform,
                                preprocess_fn=preprocess_fn,
                                knn_kw = {'metric': 'euclidean'},
                                fig_kw={'figwidth': 12, 'figheight': 12})
```



Besides visualizing and understanding the prototypes importance (i.e., larger images correspond to more important prototypes), one can also understand the diversity of each class by simply displaying all their corresponding prototypes.

```
[20]: display_images(summary=summary, imgsize=1.5)
```



For example, within the current setup, we can observe that only two prototypes are required to cover the subset of the airplane instance. On the other hand, for the subset of the car instances, we need at least six prototypes. The visualization suggests that the car class has more diversity in the feature representation and implicitly requires more prototypes to cover its instances.

Train 1-KNN

As before, we train a 1-KNN classifier to verify the quality of the prototypes returned by **ProtoSelect** against a random sampling.

```
[21]: # train 1-knn classifier using the selected prototypes
X_protos = summary.data['prototypes']
y_protos = summary.data['prototype_labels']
knn_proto = KNeighborsClassifier(n_neighbors=1, metric='euclidean')
knn_proto = knn_proto.fit(X=preprocess_fn(X_protos), y=y_protos)

[22]: np.random.seed(0)
scores = []

for i in range(10):
    rand_idx = np.random.choice(len(X_train), size=len(X_protos), replace=False)
    rands, rands_labels = X_train[rand_idx], y_train[rand_idx]

    knn_rand = KNeighborsClassifier(n_neighbors=1, metric='euclidean')
    knn_rand = knn_rand.fit(X=preprocess_fn(rands), y=rands_labels)
    scores.append(knn_rand.score(preprocess_fn(X_test), y_test))

[23]: print('ProtoSelect 1-KNN accuracy: %.3f' % (knn_proto.score(preprocess_fn(X_test), y_
    ↪test)))
print('Random 1-KNN mean accuracy: %.3f' % (np.mean(scores)))

ProtoSelect 1-KNN accuracy: 0.870
Random 1-KNN mean accuracy: 0.773
```


13.1 alibi package

13.1.1 Subpackages

alibi.api package

Submodules

alibi.api.defaults module

This module defines the default metadata and data dictionaries for each explanation method. Note that the “name” field is automatically populated upon initialization of the corresponding Explainer class.

```
alibi.api.defaults.DEFAULT_DATA_ALE: dict = {'ale0': [], 'ale_values': [],  
'constant_value': None, 'feature_deciles': None, 'feature_names': None, 'feature_values':  
[], 'target_names': None}
```

Default ALE data.

```
alibi.api.defaults.DEFAULT_DATA_ANCHOR: dict = {'anchor': [], 'coverage': None,  
'precision': None, 'raw': None}
```

Default anchor data.

```
alibi.api.defaults.DEFAULT_DATA_ANCHOR_IMG: dict = {'anchor': [], 'coverage': None,  
'precision': None, 'raw': None, 'segments': None}
```

Default anchor image data.

```
alibi.api.defaults.DEFAULT_DATA_CEM: dict = {'PN': None, 'PN_pred': None, 'PP': None,  
'PP_pred': None, 'X': None, 'X_pred': None, 'grads_graph': None, 'grads_num': None}
```

Default CEM data.

```
alibi.api.defaults.DEFAULT_DATA_CF: dict = {'all': [], 'cf': None, 'orig_class': None,  
'orig_proba': None, 'success': None}
```

Default counterfactual data.

```
alibi.api.defaults.DEFAULT_DATA_CFP: dict = {'all': [], 'cf': None, 'id_proto': None,  
'orig_class': None, 'orig_proba': None}
```

Default counterfactual prototype metadata.

```
alibi.api.defaults.DEFAULT_DATA_CFRL: dict = {'cf': None, 'condition': None, 'orig':  
None, 'target': None}
```

Default CounterfactualRL data.

```
alibi.api.defaults.DEFAULT_DATA_INTGRAD: dict = {'X': None, 'attributions': None,  
'baselines': None, 'deltas': None, 'forward_kwargs': None, 'predictions': None}
```

Default IntegratedGradients data.

```
alibi.api.defaults.DEFAULT_DATA_KERNEL_SHAP: dict = {'categorical_names': {},  
'expected_value': [], 'feature_names': [], 'raw': {'importances': {}, 'instances': None,  
'prediction': None, 'raw_prediction': None}, 'shap_values': []}
```

Default KernelShap data.

```
alibi.api.defaults.DEFAULT_DATA_PD: dict = {'feature_deciles': None, 'feature_names':  
None, 'feature_values': None, 'ice_values': None, 'pd_values': None}
```

Default PartialDependence data.

```
alibi.api.defaults.DEFAULT_DATA_PDVARIANCE: dict = {'conditional_importance': None,  
'conditional_importance_values': None, 'feature_deciles': None, 'feature_importance':  
None, 'feature_interaction': None, 'feature_names': None, 'feature_values': None,  
'pd_values': None}
```

Default PartialDependenceVariance data.

```
alibi.api.defaults.DEFAULT_DATA_PERMUTATION_IMPORTANCE: dict = {'feature_importance':  
None, 'feature_names': None, 'metric_names': None}
```

Default PermutationImportance data.

```
alibi.api.defaults.DEFAULT_DATA_PROTOSELECT: dict = {'prototype_indices': None,  
'prototype_labels': None, 'prototypes': None}
```

Default ProtoSelect data.

```
alibi.api.defaults.DEFAULT_DATA_SIM: dict = {'least_similar': None, 'most_similar': None,  
'ordered_indices': None, 'scores': None}
```

Default SimilarityExplainer data.

```
alibi.api.defaults.DEFAULT_DATA_TREE_SHAP: dict = {'categorical_names': {},  
'expected_value': [], 'feature_names': [], 'raw': {'importances': {}, 'instances': None,  
'labels': None, 'loss': None, 'prediction': None, 'raw_prediction': None},  
'shap_interaction_values': [], 'shap_values': []}
```

Default TreeShap data.

```
alibi.api.defaults.DEFAULT_META_ALE: dict = {'explanations': ['global'], 'name': None,  
'params': {}, 'type': ['blackbox'], 'version': None}
```

Default ALE metadata.

```
alibi.api.defaults.DEFAULT_META_ANCHOR: dict = {'explanations': ['local'], 'name': None,  
'params': {}, 'type': ['blackbox'], 'version': None}
```

Default anchor metadata.

```
alibi.api.defaults.DEFAULT_META_CEM: dict = {'explanations': ['local'], 'name': None,  
'params': {}, 'type': ['blackbox', 'tensorflow', 'keras'], 'version': None}
```

Default CEM metadata.

```
alibi.api.defaults.DEFAULT_META_CF: dict = {'explanations': ['local'], 'name': None,  
'params': {}, 'type': ['blackbox', 'tensorflow', 'keras'], 'version': None}
```

Default counterfactual metadata.

```
alibi.api.defaults.DEFAULT_META_CFP: dict = {'explanations': ['local'], 'name': None,
'params': {}, 'type': ['blackbox', 'tensorflow', 'keras'], 'version': None}
```

Default counterfactual prototype metadata.

```
alibi.api.defaults.DEFAULT_META_CFRL: dict = {'explanations': ['local'], 'name': None,
'params': {}, 'type': ['blackbox'], 'version': None}
```

Default CounterfactualRL metadata.

```
alibi.api.defaults.DEFAULT_META_INTGRAD: dict = {'explanations': ['local'], 'name': None,
'params': {}, 'type': ['whitebox'], 'version': None}
```

Default IntegratedGradients metadata.

```
alibi.api.defaults.DEFAULT_META_KERNEL_SHAP: dict = {'explanations': ['local', 'global'],
'name': None, 'params': {'group_names': None, 'grouped': None, 'groups': None, 'kwargs':
None, 'link': None, 'summarise_background': None, 'summarise_result': None, 'transpose':
None, 'weights': None}, 'task': None, 'type': ['blackbox'], 'version': None}
```

Default KernelShap metadata.

```
alibi.api.defaults.DEFAULT_META_PD: dict = {'explanations': ['global'], 'name': None,
'params': {}, 'type': ['blackbox'], 'version': None}
```

Default PartialDependence metadata.

```
alibi.api.defaults.DEFAULT_META_PDVARIANCE: dict = {'explanations': ['global'], 'name':
None, 'params': {}, 'type': ['blackbox'], 'version': None}
```

Default PartialDependenceVariance metadata.

```
alibi.api.defaults.DEFAULT_META_PERMUTATION_IMPORTANCE: dict = {'explanations':
['global'], 'name': None, 'params': {}, 'type': ['blackbox'], 'version': None}
```

Default PermutationImportance metadata.

```
alibi.api.defaults.DEFAULT_META_PROTOSSELECT: dict = {'explanation': ['global'], 'name':
None, 'params': {}, 'type': ['data'], 'version': None}
```

Default ProtoSelect metadata.

```
alibi.api.defaults.DEFAULT_META_SIM: dict = {'explanations': ['local'], 'name': None,
'params': {}, 'type': ['whitebox'], 'version': None}
```

Default SimilarityExplainer metadata.

```
alibi.api.defaults.DEFAULT_META_TREE_SHAP: dict = {'explanations': ['local', 'global'],
'name': None, 'params': {'algorithm': None, 'approximate': None, 'explain_loss': None,
'interactions': None, 'kwargs': None, 'model_output': None, 'summarise_background': None,
'summarise_result': None}, 'task': None, 'type': ['whitebox'], 'version': None}
```

Default TreeShap metadata.

```
alibi.api.defaults.KERNEL_SHAP_PARAMS = ['link', 'group_names', 'grouped', 'groups',
'weights', 'summarise_background', 'summarise_result', 'transpose', 'kwargs']
```

KernelShap parameters updated and returned in metadata['params']. See [alibi.explainers.shap_wrappers.KernelShap](#).

```
alibi.api.defaults.TREE_SHAP_PARAMS = ['model_output', 'summarise_background',
'summarise_result', 'approximate', 'interactions', 'explain_loss', 'algorithm', 'kwargs']
```

TreeShap parameters updated and returned in metadata['params']. See [alibi.explainers.shap_wrappers.TreeShap](#).

alibi.api.interfaces module

class alibi.api.interfaces.**AlibiPrettyPrinter**(*args, **kwargs)

Bases: `PrettyPrinter`

Overrides the built in dictionary pretty representation to look more similar to the external prettyprinter library.

class alibi.api.interfaces.**Base**(meta=_Nothing.NOTHING)

Bases: `object`

Base class for all *alibi* algorithms. Implements a structured approach to handle metadata.

meta: `dict`

Object metadata.

class alibi.api.interfaces.**Explainer**(meta=_Nothing.NOTHING)

Bases: `ABC`, `Base`

Base class for explainer algorithms from *alibi.explainers*.

abstract explain(X)

Return type

`Explanation`

classmethod load(path, predictor)

Load an explainer from disk.

Parameters

- **path** (`Union[str, PathLike]`) – Path to a directory containing the saved explainer.
- **predictor** (`Any`) – Model or prediction function used to originally initialize the explainer.

Return type

`Explainer`

Returns

An explainer instance.

reset_predictor(predictor)

Resets the predictor.

Parameters

predictor (`Any`) – New predictor.

Return type

`None`

save(path)

Save an explainer to disk. Uses the *dill* module.

Parameters

path (`Union[str, PathLike]`) – Path to a directory. A new directory will be created if one does not exist.

Return type

`None`

class alibi.api.interfaces.**Explanation**(meta, data)

Bases: `object`

Explanation class returned by explainers.

__attrs_post_init__()

Expose keys stored in *self.meta* and *self.data* as attributes of the class.

__getitem__(item)

This method is purely for deprecating previous behaviour of accessing explanation data via items in the returned dictionary.

data: `dict`

classmethod from_json(jsonrepr)

Create an instance of an *Explanation* class using a *json* representation of the *Explanation*.

Parameters

jsonrepr – *json* representation of an explanation.

Return type

Explanation

Returns

An *Explanation* object.

meta: `dict`

to_json()

Serialize the explanation data and metadata into a *json* format.

Return type

`str`

Returns

String containing *json* representation of the explanation.

class alibi.api.interfaces.FitMixin

Bases: `ABC`

abstract fit(X)

Return type

Explainer

class alibi.api.interfaces.Summariser(meta=_Nothing.NOTHING)

Bases: `ABC`, *Base*

Base class for prototype algorithms from *alibi.prototypes*.

classmethod load(path)

Return type

Summariser

save(path)

Return type

`None`

abstract summarise(num_prototypes)

Return type

Explanation

`alibi.api.interfaces.default_meta()`

Return type

`dict`

alibi.confidence package

The ‘alibi.confidence’ module includes trust scores.

```
class alibi.confidence.LinearityMeasure(method='grid', epsilon=0.04, nb_samples=10, res=100,
                                         alphas=None, model_type='classifier', agg='pairwise',
                                         verbose=False)
```

Bases: `object`

```
__init__(method='grid', epsilon=0.04, nb_samples=10, res=100, alphas=None, model_type='classifier',
          agg='pairwise', verbose=False)
```

Parameters

- **method** (`str`) – Method for sampling. Supported methods: 'knn' | 'grid'.
- **epsilon** (`float`) – Size of the sampling region around the central instance as a percentage of the features range.
- **nb_samples** (`int`) – Number of samples to generate.
- **res** (`int`) – Resolution of the grid. Number of intervals in which the feature range is discretized.
- **alphas** (`Optional`[`ndarray`]) – Coefficients in the superposition.
- **agg** (`str`) – Aggregation method. Supported values: 'global' | 'pairwise'.
- **model_type** (`str`) – Type of task. Supported values: 'regressor' | 'classifier'.

fit(*X_train*)

Parameters

X_train (`ndarray`) – Training set.

Return type

`None`

score(*predict_fn*, *x*)

Parameters

- **predict_fn** (`Callable`) – Prediction function.
- **x** (`ndarray`) – Instance of interest.

Return type

`ndarray`

Returns

Linearity measure.

```
class alibi.confidence.TrustScore(k_filter=10, alpha=0.0, filter_type=None, leaf_size=40,
                                  metric='euclidean', dist_filter_type='point')
```

Bases: `object`

```
__init__(k_filter=10, alpha=0.0, filter_type=None, leaf_size=40, metric='euclidean',
         dist_filter_type='point')
```

Initialize trust scores.

Parameters

- **k_filter** (`int`) – Number of neighbors used during either kNN distance or probability filtering.
- **alpha** (`float`) – Fraction of instances to filter out to reduce impact of outliers.
- **filter_type** (`Optional[str]`) – Filter method: 'distance_knn' | 'probability_knn'.
- **leaf_size** (`int`) – Number of points at which to switch to brute-force. Affects speed and memory required to build trees. Memory to store the tree scales with $n_samples / leaf_size$.
- **metric** (`str`) – Distance metric used for the tree. See *sklearn* DistanceMetric class for a list of available metrics.
- **dist_filter_type** (`str`) – Use either the distance to the k-nearest point (`dist_filter_type = 'point'`) or the average distance from the first to the k-nearest point in the data (`dist_filter_type = 'mean'`).

`filter_by_distance_knn(X)`

Filter out instances with low kNN density. Calculate distance to k-nearest point in the data for each instance and remove instances above a cutoff distance.

Parameters

X (`ndarray`) – Data.

Return type

`ndarray`

Returns

Filtered data.

`filter_by_probability_knn(X, Y)`

Filter out instances with high label disagreement amongst its k nearest neighbors.

Parameters

- **X** (`ndarray`) – Data.
- **Y** (`ndarray`) – Predicted class labels.

Return type

`Tuple[ndarray, ndarray]`

Returns

Filtered data and labels.

`fit(X, Y, classes=None)`

Build KDTrees for each prediction class.

Parameters

- **X** (`ndarray`) – Data.
- **Y** (`ndarray`) – Target labels, either one-hot encoded or the actual class label.
- **classes** (`Optional[int]`) – Number of prediction classes, needs to be provided if *Y* equals the predicted class.

Return type`None`**score**(*X*, *Y*, *k*=2, *dist_type*='point')

Calculate trust scores = ratio of distance to closest class other than the predicted class to distance to predicted class.

Parameters

- **X** (`ndarray`) – Instances to calculate trust score for.
- **Y** (`ndarray`) – Either prediction probabilities for each class or the predicted class.
- **k** (`int`) – Number of nearest neighbors used for distance calculation.
- **dist_type** (`str`) – Use either the distance to the k-nearest point (`dist_type = 'point'`) or the average distance from the first to the k-nearest point in the data (`dist_type = 'mean'`).

Return type`Tuple[ndarray, ndarray]`**Returns**

Batch with trust scores and the closest not predicted class.

alibi.confidence.linearity_measure(*predict_fn*, *x*, *feature_range*=None, *method*='grid', *X_train*=None, *epsilon*=0.04, *nb_samples*=10, *res*=100, *alphas*=None, *agg*='global', *model_type*='classifier')

Calculate the linearity measure of the model around an instance of interest *x*.

Parameters

- **predict_fn** (`Callable`) – Predict function.
- **x** (`ndarray`) – Instance of interest.
- **feature_range** (`Union[List, ndarray, None]`) – Array with min and max values for each feature.
- **method** (`str`) – Method for sampling. Supported values: 'knn' | 'grid'.
- **X_train** (`Optional[ndarray]`) – Training set.
- **epsilon** (`float`) – Size of the sampling region as a percentage of the feature range.
- **nb_samples** (`int`) – Number of samples to generate.
- **res** (`int`) – Resolution of the grid. Number of intervals in which the features range is discretized.
- **alphas** (`Optional[ndarray]`) – Coefficients in the superposition.
- **agg** (`str`) – Aggregation method. Supported values: 'global' | 'pairwise'.
- **model_type** (`str`) – Type of task. Supported values: 'regressor' | 'classifier'.

Return type`ndarray`**Returns**

Linearity measure.

Submodules

alibi.confidence.model_linearity module

```
class alibi.confidence.model_linearity.LinearityMeasure(method='grid', epsilon=0.04,
                                                         nb_samples=10, res=100, alphas=None,
                                                         model_type='classifier', agg='pairwise',
                                                         verbose=False)
```

Bases: `object`

```
__init__(method='grid', epsilon=0.04, nb_samples=10, res=100, alphas=None, model_type='classifier',
         agg='pairwise', verbose=False)
```

Parameters

- **method** (`str`) – Method for sampling. Supported methods: 'knn' | 'grid'.
- **epsilon** (`float`) – Size of the sampling region around the central instance as a percentage of the features range.
- **nb_samples** (`int`) – Number of samples to generate.
- **res** (`int`) – Resolution of the grid. Number of intervals in which the feature range is discretized.
- **alphas** (`Optional`[`ndarray`]) – Coefficients in the superposition.
- **agg** (`str`) – Aggregation method. Supported values: 'global' | 'pairwise'.
- **model_type** (`str`) – Type of task. Supported values: 'regressor' | 'classifier'.

fit(*X_train*)

Parameters

X_train (`ndarray`) – Training set.

Return type

`None`

score(*predict_fn*, *x*)

Parameters

- **predict_fn** (`Callable`) – Prediction function.
- **x** (`ndarray`) – Instance of interest.

Return type

`ndarray`

Returns

Linearity measure.

```
alibi.confidence.model_linearity.infer_feature_range(X_train)
```

Infers the feature range from the training set.

Parameters

X_train (`ndarray`) – Training set.

Return type

`ndarray`

Returns

Feature range.

```
alibi.confidence.model_linearity.linearity_measure(predict_fn, x, feature_range=None,
                                                  method='grid', X_train=None, epsilon=0.04,
                                                  nb_samples=10, res=100, alphas=None,
                                                  agg='global', model_type='classifier')
```

Calculate the linearity measure of the model around an instance of interest x.

Parameters

- **predict_fn** (`Callable`) – Predict function.
- **x** (`ndarray`) – Instance of interest.
- **feature_range** (`Union[List, ndarray, None]`) – Array with min and max values for each feature.
- **method** (`str`) – Method for sampling. Supported values: 'knn' | 'grid'.
- **X_train** (`Optional[ndarray]`) – Training set.
- **epsilon** (`float`) – Size of the sampling region as a percentage of the feature range.
- **nb_samples** (`int`) – Number of samples to generate.
- **res** (`int`) – Resolution of the grid. Number of intervals in which the features range is discretized.
- **alphas** (`Optional[ndarray]`) – Coefficients in the superposition.
- **agg** (`str`) – Aggregation method. Supported values: 'global' | 'pairwise'.
- **model_type** (`str`) – Type of task. Supported values: 'regressor' | 'classifier'.

Return type

`ndarray`

Returns

Linearity measure.

alibi.confidence.trustscore module

```
class alibi.confidence.trustscore.TrustScore(k_filter=10, alpha=0.0, filter_type=None, leaf_size=40,
                                             metric='euclidean', dist_filter_type='point')
```

Bases: `object`

```
__init__(k_filter=10, alpha=0.0, filter_type=None, leaf_size=40, metric='euclidean',
         dist_filter_type='point')
```

Initialize trust scores.

Parameters

- **k_filter** (`int`) – Number of neighbors used during either kNN distance or probability filtering.
- **alpha** (`float`) – Fraction of instances to filter out to reduce impact of outliers.
- **filter_type** (`Optional[str]`) – Filter method: 'distance_knn' | 'probability_knn'.
- **leaf_size** (`int`) – Number of points at which to switch to brute-force. Affects speed and memory required to build trees. Memory to store the tree scales with $n_samples / leaf_size$.

- **metric** (`str`) – Distance metric used for the tree. See *sklearn* DistanceMetric class for a list of available metrics.
- **dist_filter_type** (`str`) – Use either the distance to the k-nearest point (`dist_filter_type = 'point'`) or the average distance from the first to the k-nearest point in the data (`dist_filter_type = 'mean'`).

filter_by_distance_knn(*X*)

Filter out instances with low kNN density. Calculate distance to k-nearest point in the data for each instance and remove instances above a cutoff distance.

Parameters

X (`ndarray`) – Data.

Return type

`ndarray`

Returns

Filtered data.

filter_by_probability_knn(*X*, *Y*)

Filter out instances with high label disagreement amongst its k nearest neighbors.

Parameters

- **X** (`ndarray`) – Data.
- **Y** (`ndarray`) – Predicted class labels.

Return type

`Tuple[ndarray, ndarray]`

Returns

Filtered data and labels.

fit(*X*, *Y*, *classes=None*)

Build KDTrees for each prediction class.

Parameters

- **X** (`ndarray`) – Data.
- **Y** (`ndarray`) – Target labels, either one-hot encoded or the actual class label.
- **classes** (`Optional[int]`) – Number of prediction classes, needs to be provided if *Y* equals the predicted class.

Return type

`None`

score(*X*, *Y*, *k=2*, *dist_type='point'*)

Calculate trust scores = ratio of distance to closest class other than the predicted class to distance to predicted class.

Parameters

- **X** (`ndarray`) – Instances to calculate trust score for.
- **Y** (`ndarray`) – Either prediction probabilities for each class or the predicted class.
- **k** (`int`) – Number of nearest neighbors used for distance calculation.
- **dist_type** (`str`) – Use either the distance to the k-nearest point (`dist_type = 'point'`) or the average distance from the first to the k-nearest point in the data (`dist_type = 'mean'`).

Return type`Tuple[ndarray, ndarray]`**Returns***Batch with trust scores and the closest not predicted class.***alibi.datasets package**`alibi.datasets.fetch_adult(features_drop=None, return_X_y=False, url_id=0)`

Downloads and pre-processes ‘adult’ dataset. More info: <http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/>

Parameters

- **features_drop** (`Optional[list]`) – List of features to be dropped from dataset, by default drops ["fnlwgt", "Education-Num"].
- **return_X_y** (`bool`) – If True, return features *X* and labels *y* as *numpy* arrays. If False return a *Bunch* object.
- **url_id** (`int`) – Index specifying which URL to use for downloading.

Return type`Union[Bunch, Tuple[ndarray, ndarray]]`**Returns**

- *Bunch* – Dataset, labels, a list of features and a dictionary containing a list with the potential categories for each categorical feature where the key refers to the feature column.
- (*data*, *target*) – Tuple if `return_X_y=True`

`alibi.datasets.fetch_fashion_mnist(return_X_y=False)`

Loads the Fashion MNIST dataset.

Parameters

return_X_y (`bool`) – If True, an $N \times M \times P$ array of data points and *N*-array of labels are returned instead of a dict.

Return type`Union[Bunch, Tuple[ndarray, ndarray]]`**Returns**

- If `return_X_y=False`, a *Bunch* object with fields ‘data’, ‘targets’ and ‘target_names’
- *is returned. Otherwise an array with data points and an array of labels is returned.*

`alibi.datasets.fetch_imagenet(category='Persian cat', nb_images=10, target_size=(299, 299), min_std=10.0, seed=42, return_X_y=False)`**Return type**`None``alibi.datasets.fetch_imagenet_10(url_id=0)`

Sample dataset extracted from imagenet in a dictionary format. The train set contains 1000 random samples, 100 for each of the following 10 selected classes:

- stingray
- trilobite
- centipede

- slug
- snail
- Rhodesian ridgeback
- beagle
- golden retriever
- sea lion
- espresso

The test set contains 50 random samples, 5 for each of the classes above.

Parameters

url_id (`int`) – Index specifying which URL to use for downloading.

Return type

`Dict`

Returns

Dictionary with the following keys –

- `trainset` - train set tuple (`X_train`, `y_train`)
- `testset` - test set tuple (`X_test`, `y_test`)
- `int_to_str_labels` - map from target to target name
- `str_to_int_labels` - map from target name to target

`alibi.datasets.fetch_movie_sentiment(return_X_y=False, url_id=0)`

The movie review dataset, equally split between negative and positive reviews.

Parameters

- **return_X_y** (`bool`) – If `True`, return features `X` and labels `y` as *Python* lists. If `False` return a *Bunch* object.
- **url_id** (`int`) – Index specifying which URL to use for downloading

Return type

`Union[Bunch, Tuple[list, list]]`

Returns

- *Bunch* – Movie reviews and sentiment labels (0 means ‘negative’ and 1 means ‘positive’).
- (`data`, `target`) – Tuple if `return_X_y=True`.

`alibi.datasets.load_cats(target_size=(299, 299), return_X_y=False)`

A small sample of Imagenet-like public domain images of cats used primarily for examples. The images were hand-collected using flickr.com by searching for various cat types, filtered by images in the public domain.

Parameters

- **target_size** (`tuple`) – Size of the returned images, used to crop images for a specified model input size.
- **return_X_y** (`bool`) – If `True`, return features `X` and labels `y` as *numpy* arrays. If `False` return a *Bunch* object

Return type

`Union[Bunch, Tuple[ndarray, ndarray]]`

Returns

- *Bunch* – Bunch object with fields ‘data’, ‘target’ and ‘target_names’. Both *targets* and *target_names* are taken from the original Imagenet.
- *(data, target)* – Tuple if `return_X_y=True`.

Submodules

alibi.datasets.default module

`alibi.datasets.default.fetch_adult(features_drop=None, return_X_y=False, url_id=0)`

Downloads and pre-processes ‘adult’ dataset. More info: <http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/>

Parameters

- **features_drop** (`Optional[list]`) – List of features to be dropped from dataset, by default drops ["fnlwgt", "Education-Num"].
- **return_X_y** (`bool`) – If `True`, return features *X* and labels *y* as *numpy* arrays. If `False` return a *Bunch* object.
- **url_id** (`int`) – Index specifying which URL to use for downloading.

Return type

`Union[Bunch, Tuple[ndarray, ndarray]]`

Returns

- *Bunch* – Dataset, labels, a list of features and a dictionary containing a list with the potential categories for each categorical feature where the key refers to the feature column.
- *(data, target)* – Tuple if `return_X_y=True`

`alibi.datasets.default.fetch_imagenet(category='Persian cat', nb_images=10, target_size=(299, 299), min_std=10.0, seed=42, return_X_y=False)`

Return type

`None`

`alibi.datasets.default.fetch_imagenet_10(url_id=0)`

Sample dataset extracted from imagenet in a dictionary format. The train set contains 1000 random samples, 100 for each of the following 10 selected classes:

- stingray
- trilobite
- centipede
- slug
- snail
- Rhodesian ridgeback
- beagle
- golden retriever
- sea lion

- espresso

The test set contains 50 random samples, 5 for each of the classes above.

Parameters

url_id (`int`) – Index specifying which URL to use for downloading.

Return type

`Dict`

Returns

Dictionary with the following keys –

- `trainset` - train set tuple (`X_train`, `y_train`)
- `testset` - test set tuple (`X_test`, `y_test`)
- `int_to_str_labels` - map from target to target name
- `str_to_int_labels` - map from target name to target

`alibi.datasets.default.fetch_movie_sentiment(return_X_y=False, url_id=0)`

The movie review dataset, equally split between negative and positive reviews.

Parameters

- **return_X_y** (`bool`) – If `True`, return features `X` and labels `y` as *Python* lists. If `False` return a *Bunch* object.
- **url_id** (`int`) – Index specifying which URL to use for downloading

Return type

`Union[Bunch, Tuple[list, list]]`

Returns

- *Bunch* – Movie reviews and sentiment labels (0 means ‘negative’ and 1 means ‘positive’).
- (`data`, `target`) – Tuple if `return_X_y=True`.

`alibi.datasets.default.load_cats(target_size=(299, 299), return_X_y=False)`

A small sample of Imagenet-like public domain images of cats used primarily for examples. The images were hand-collected using flickr.com by searching for various cat types, filtered by images in the public domain.

Parameters

- **target_size** (`tuple`) – Size of the returned images, used to crop images for a specified model input size.
- **return_X_y** (`bool`) – If `True`, return features `X` and labels `y` as *numpy* arrays. If `False` return a *Bunch* object

Return type

`Union[Bunch, Tuple[ndarray, ndarray]]`

Returns

- *Bunch* – Bunch object with fields ‘data’, ‘target’ and ‘target_names’. Both *targets* and *target_names* are taken from the original Imagenet.
- (`data`, `target`) – Tuple if `return_X_y=True`.

alibi.datasets.tensorflow module

`alibi.datasets.tensorflow.fetch_fashion_mnist(return_X_y=False)`

Loads the Fashion MNIST dataset.

Parameters

return_X_y (`bool`) – If `True`, an $N \times M \times P$ array of data points and N -array of labels are returned instead of a dict.

Return type

`Union[Bunch, Tuple[ndarray, ndarray]]`

Returns

- If `return_X_y=False`, a `Bunch` object with fields ‘data’, ‘targets’ and ‘target_names’
- *is returned. Otherwise an array with data points and an array of labels is returned.*

alibi.explainers package

The ‘alibi.explainers’ module includes feature importance, counterfactual and anchor-based explainers.

```
class alibi.explainers.ALE(predictor, feature_names=None, target_names=None,
                           check_feature_resolution=True, low_resolution_threshold=10,
                           extrapolate_constant=True, extrapolate_constant_perc=10.0,
                           extrapolate_constant_min=0.1)
```

Bases: [Explainer](#)

```
__init__(predictor, feature_names=None, target_names=None, check_feature_resolution=True,
          low_resolution_threshold=10, extrapolate_constant=True, extrapolate_constant_perc=10.0,
          extrapolate_constant_min=0.1)
```

Accumulated Local Effects for tabular datasets. Current implementation supports first order feature effects of numerical features.

Parameters

- **predictor** (`Callable[[ndarray], ndarray]`) – A callable that takes in an $N \times F$ array as input and outputs an $N \times T$ array (N - number of data points, F - number of features, T - number of outputs/targets (e.g. 1 for single output regression, ≥ 2 for classification)).
- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.
- **target_names** (`Optional[List[str]]`) – A list of target/output names used for displaying results.
- **check_feature_resolution** (`bool`) – If `True`, the number of unique values is calculated for each feature and if it is less than `low_resolution_threshold` then the feature values are used for grid-points instead of quantiles. This may increase the runtime of the algorithm for large datasets. Only used for features without custom grid-points specified in [alibi.explainers.ale.ALE.explain\(\)](#).
- **low_resolution_threshold** (`int`) – If a feature has at most this many unique values, these are used as the grid points instead of quantiles. This is to avoid situations when the quantile algorithm returns quantiles between discrete values which can result in jumps in the ALE plot obscuring the true effect. Only used if `check_feature_resolution` is `True` and for features without custom grid-points specified in [alibi.explainers.ale.ALE.explain\(\)](#).

- **extrapolate_constant** (`bool`) – If a feature is constant, only one quantile exists where all the data points lie. In this case the ALE value at that point is zero, however this may be misleading if the feature does have an effect on the model. If this parameter is set to `True`, the ALE values are calculated on an interval surrounding the constant value. The interval length is controlled by the `extrapolate_constant_perc` and `extrapolate_constant_min` arguments.
- **extrapolate_constant_perc** (`float`) – Percentage by which to extrapolate a constant feature value to create an interval for ALE calculation. If q is the constant feature value, creates an interval $[q - q/\text{extrapolate_constant_perc}, q + q/\text{extrapolate_constant_perc}]$ for which ALE is calculated. Only relevant if `extrapolate_constant` is set to `True`.
- **extrapolate_constant_min** (`float`) – Controls the minimum extrapolation length for constant features. An interval constructed for constant features is guaranteed to be $2 \times \text{extrapolate_constant_min}$ wide centered on the feature value. This allows for capturing model behaviour around constant features which have small value so that `extrapolate_constant_perc` is not so helpful. Only relevant if `extrapolate_constant` is set to `True`.

explain(*X*, features=None, min_bin_points=4, grid_points=None)

Calculate the ALE curves for each feature with respect to the dataset *X*.

Parameters

- **X** (`ndarray`) – An $N \times F$ tabular dataset used to calculate the ALE curves. This is typically the training dataset or a representative sample.
- **features** (`Optional[List[int]]`) – Features for which to calculate ALE.
- **min_bin_points** (`int`) – Minimum number of points each discretized interval should contain to ensure more precise ALE estimation. Only relevant for adaptive grid points (i.e., features without an entry in the `grid_points` dictionary).
- **grid_points** (`Optional[Dict[int, ndarray]]`) – Custom grid points. Must be a *dict* where the keys are features indices and the values are monotonically increasing *numpy* arrays defining the grid points for each feature. See the [Notes](#) section for the default behavior when potential edge-cases arise when using grid-points. If no grid points are specified (i.e. the feature is missing from the `grid_points` dictionary), deciles discretization is used instead.

Return type

[Explanation](#)

Returns

explanation – An *Explanation* object containing the data and the metadata of the calculated ALE curves. See usage at [ALE examples](#) for details.

Notes

Consider f to be a feature of interest. We denote possible feature values of f by X (i.e. the values from the dataset column corresponding to feature f), by O a user-specified grid-point value, and by $(X|O)$ an overlap between a grid-point and a feature value. We can encounter the following edge-cases:

- Grid points outside the feature range. Consider the following example: $O O O X X O X O X O O$, where 3 grid-points are smaller than the minimum value in f , and 2 grid-points are larger than the maximum value in f . The empty leading and ending bins are removed. The grid-points considered

will be: $O X X O X O X O$.

- Grid points that do not cover the entire feature range. Consider the following example: $X\ X\ O\ X\ X\ O\ X\ O\ X\ X\ X\ X$. Two auxiliary grid-points are added which correspond the value of the minimum and maximum value of feature f . The grid-points considered will be: $(O|X)\ X\ O\ X\ X\ O\ X\ O\ X\ X\ X\ X\ (X|O)$.
- Grid points that do not contain any values in between. Consider the following example: $(O|X)\ X\ X\ O\ O\ X\ O\ X\ O\ O\ (X|O)$. The intervals which do not contain any feature values are removed/merged. The grid-points considered will be: $(O|X)\ X\ X\ O\ X\ O\ X\ O\ (X|O)$.

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (*Callable*) – New predictor function.

Return type

None

```
class alibi.explainers.AnchorImage(predictor, image_shape, dtype=<class 'numpy.float32'>,
                                   segmentation_fn='slic', segmentation_kwargs=None,
                                   images_background=None, seed=None)
```

Bases: *Explainer*

```
__init__(predictor, image_shape, dtype=<class 'numpy.float32'>, segmentation_fn='slic',
          segmentation_kwargs=None, images_background=None, seed=None)
```

Initialize anchor image explainer.

Parameters

- **predictor** (*Callable*[[*ndarray*], *ndarray*]) – A callable that takes a *numpy* array of N data points as inputs and returns N outputs.
- **image_shape** (*tuple*) – Shape of the image to be explained. The channel axis is expected to be last.
- **dtype** (*Type*[*generic*]) – A *numpy* scalar type that corresponds to the type of input array expected by *predictor*. This may be used to construct arrays of the given type to be passed through the *predictor*. For most use cases this argument should have no effect, but it is exposed for use with predictors that would break when called with an array of unsupported type.
- **segmentation_fn** (*Any*) – Any of the built in segmentation function strings: 'felzenszwalb', 'slic' or 'quickshift' or a custom segmentation function (callable) which returns an image mask with labels for each superpixel. The segmentation function is expected to return a segmentation mask containing all integer values from 0 to $K-1$, where K is the number of image segments (superpixels). See <http://scikit-image.org/docs/dev/api/skimask.segmentation.html> for more info.
- **segmentation_kwargs** (*Optional*[*dict*]) – Keyword arguments for the built in segmentation functions.
- **images_background** (*Optional*[*ndarray*]) – Images to overlay superpixels on.
- **seed** (*Optional*[*int*]) – If set, ensures different runs with the same input will yield same explanation.

Raises

- **alibi.exceptions.PredictorCallError** – If calling *predictor* fails at runtime.
- **alibi.exceptions.PredictorReturnTypeError** – If the return type of *predictor* is not *np.ndarray*.

```
explain(image, p_sample=0.5, threshold=0.95, delta=0.1, tau=0.15, batch_size=100,
        coverage_samples=10000, beam_size=1, stop_on_first=False, max_anchor_size=None,
        min_samples_start=100, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000,
        verbose=False, verbose_every=1, **kwargs)
```

Explain instance and return anchor with metadata.

Parameters

- **image** (ndarray) – Image to be explained.
- **p_sample** (float) – The probability of simulating the absence of a superpixel. If the *images_background* is not provided, the absent superpixels will be replaced by the average value of their constituent pixels. Otherwise, the synthetic instances are created by fixing the present superpixels and superimposing another image from the *images_background* over the rest of the absent superpixels.
- **threshold** (float) – Minimum anchor precision threshold. The algorithm tries to find an anchor that maximizes the coverage under precision constraint. The precision constraint is formally defined as $P(\text{prec}(A) \geq t) \geq 1 - \delta$, where A is an anchor, t is the *threshold* parameter, δ is the *delta* parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, we are seeking for an anchor having its precision greater or equal than the given *threshold* with a confidence of $(1 - \text{delta})$. A higher value guarantees that the anchors are faithful to the model, but also leads to more computation time. Note that there are cases in which the precision constraint cannot be satisfied due to the quantile-based discretisation of the numerical features. If that is the case, the best (i.e. highest coverage) non-eligible anchor is returned.
- **delta** (float) – Significance threshold. $1 - \text{delta}$ represents the confidence threshold for the anchor precision (see *threshold*) and the selection of the best anchor candidate in each iteration (see *tau*).
- **tau** (float) – Multi-armed bandit parameter used to select candidate anchors in each iteration. The multi-armed bandit algorithm tries to find within a tolerance *tau* the most promising (i.e. according to the precision) *beam_size* candidate anchor(s) from a list of proposed anchors. Formally, when the *beam_size*=1, the multi-armed bandit algorithm seeks to find an anchor A such that $P(\text{prec}(A) \geq \text{prec}(A^*) - \tau) \geq 1 - \delta$, where A^* is the anchor with the highest true precision (which we don't know), τ is the *tau* parameter, δ is the *delta* parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, in each iteration, the algorithm returns with a probability of at least $1 - \text{delta}$ an anchor A with a precision within an error tolerance of *tau* from the precision of the highest true precision anchor A^* . A bigger value for *tau* means faster convergence but also looser anchor conditions.
- **batch_size** (int) – Batch size used for sampling. The Anchor algorithm will query the black-box model in batches of size *batch_size*. A larger *batch_size* gives more confidence in the anchor, again at the expense of computation time since it involves more model prediction calls.
- **coverage_samples** (int) – Number of samples used to estimate coverage from during result search.
- **beam_size** (int) – Number of candidate anchors selected by the multi-armed bandit algorithm in each iteration from a list of proposed anchors. A bigger beam width can lead to a better overall anchor (i.e. prevents the algorithm of getting stuck in a local maximum) at the expense of more computation time.
- **stop_on_first** (bool) – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.

- **max_anchor_size** (`Optional[int]`) – Maximum number of features in result.
- **min_samples_start** (`int`) – Min number of initial samples.
- **n_covered_ex** (`int`) – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with *desired_label* are stored).
- **binary_cache_size** (`int`) – The result search pre-allocates *binary_cache_size* batches for storing the binary arrays returned during sampling.
- **cache_margin** (`int`) – When only `max(cache_margin, batch_size)` positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
- **verbose** (`bool`) – Display updates during the anchor search iterations.
- **verbose_every** (`int`) – Frequency of displayed iterations during anchor search process.

Return type

Explanation

Returns

explanation – *Explanation* object containing the anchor explaining the instance with additional metadata as attributes. See usage at [AnchorImage examples](#) for details.

generate_superpixels(*image*)

Generates superpixels from (i.e., segments) an image.

Parameters

image (`ndarray`) – A grayscale or RGB image.

Return type

`ndarray`

Returns

A $[H, W]$ array of integers. Each integer is a segment (superpixel) label.

overlay_mask(*image, segments, mask_features, scale=(0, 255)*)

Overlay image with mask described by the mask features.

Parameters

- **image** (`ndarray`) – Image to be explained.
- **segments** (`ndarray`) – Superpixels.
- **mask_features** (`list`) – List with superpixels present in mask.
- **scale** (`tuple`) – Pixel scale for masked image.

Return type

`ndarray`

Returns

masked_image – Image overlaid with mask.

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (`Callable`) – New predictor function.

Return type

None

class `alibi.explainers.AnchorTabular`(*predictor*, *feature_names*, *categorical_names*=None, *dtype*=<class 'numpy.float32'>, *ohe*=False, *seed*=None)

Bases: [Explainer](#), [FitMixin](#)

__init__(*predictor*, *feature_names*, *categorical_names*=None, *dtype*=<class 'numpy.float32'>, *ohe*=False, *seed*=None)

Parameters

- **predictor** ([Callable](#)[[[ndarray](#)], [ndarray](#)]) – A callable that takes a *numpy* array of *N* data points as inputs and returns *N* outputs.
- **feature_names** ([List](#)[[str](#)]) – List with feature names.
- **categorical_names** ([Optional](#)[[Dict](#)[[int](#), [List](#)[[str](#)]]]) – Dictionary where keys are feature columns and values are the categories for the feature.
- **dtype** ([Type](#)[[generic](#)]) – A *numpy* scalar type that corresponds to the type of input array expected by *predictor*. This may be used to construct arrays of the given type to be passed through the *predictor*. For most use cases this argument should have no effect, but it is exposed for use with predictors that would break when called with an array of unsupported type.
- **ohe** ([bool](#)) – Whether the categorical variables are one-hot encoded (OHE) or not. If not OHE, they are assumed to have ordinal encodings.
- **seed** ([Optional](#)[[int](#)]) – Used to set the random number generator for repeatability purposes.

Raises

- [alibi.exceptions.PredictorCallError](#) – If calling *predictor* fails at runtime.
- [alibi.exceptions.PredictorReturnTypeError](#) – If the return type of *predictor* is not *np.ndarray*.

add_names_to_exp(*explanation*)

Add feature names to explanation dictionary.

Parameters

explanation ([dict](#)) – Dict with anchors and additional metadata.

Return type

None

explain(*X*, *threshold*=0.95, *delta*=0.1, *tau*=0.15, *batch_size*=100, *coverage_samples*=10000, *beam_size*=1, *stop_on_first*=False, *max_anchor_size*=None, *min_samples_start*=100, *n_covered_ex*=10, *binary_cache_size*=10000, *cache_margin*=1000, *verbose*=False, *verbose_every*=1, ***kwargs*)

Explain prediction made by classifier on instance *X*.

Parameters

- **X** ([ndarray](#)) – Instance to be explained.
- **threshold** ([float](#)) – Minimum anchor precision threshold. The algorithm tries to find an anchor that maximizes the coverage under precision constraint. The precision constraint is formally defined as $P(\text{prec}(A) \geq t) \geq 1 - \delta$, where *A* is an anchor, *t* is the *threshold* parameter, δ is the *delta* parameter, and *prec*(·) denotes the precision of an anchor. In other words, we are seeking for an anchor having its precision greater or equal than the given

threshold with a confidence of $(1 - \delta)$. A higher value guarantees that the anchors are faithful to the model, but also leads to more computation time. Note that there are cases in which the precision constraint cannot be satisfied due to the quantile-based discretisation of the numerical features. If that is the case, the best (i.e. highest coverage) non-eligible anchor is returned.

- **delta** (`float`) – Significance threshold. $1 - \delta$ represents the confidence threshold for the anchor precision (see *threshold*) and the selection of the best anchor candidate in each iteration (see *tau*).
- **tau** (`float`) – Multi-armed bandit parameter used to select candidate anchors in each iteration. The multi-armed bandit algorithm tries to find within a tolerance *tau* the most promising (i.e. according to the precision) *beam_size* candidate anchor(s) from a list of proposed anchors. Formally, when the *beam_size*=1, the multi-armed bandit algorithm seeks to find an anchor *A* such that $P(\text{prec}(A) \geq \text{prec}(A^*) - \tau) \geq 1 - \delta$, where *A** is the anchor with the highest true precision (which we don't know), τ is the *tau* parameter, δ is the *delta* parameter, and *prec*(·) denotes the precision of an anchor. In other words, in each iteration, the algorithm returns with a probability of at least $1 - \delta$ an anchor *A* with a precision within an error tolerance of *tau* from the precision of the highest true precision anchor *A**. A bigger value for *tau* means faster convergence but also looser anchor conditions.
- **batch_size** (`int`) – Batch size used for sampling. The Anchor algorithm will query the black-box model in batches of size *batch_size*. A larger *batch_size* gives more confidence in the anchor, again at the expense of computation time since it involves more model prediction calls.
- **coverage_samples** (`int`) – Number of samples used to estimate coverage from during result search.
- **beam_size** (`int`) – Number of candidate anchors selected by the multi-armed bandit algorithm in each iteration from a list of proposed anchors. A bigger beam width can lead to a better overall anchor (i.e. prevents the algorithm of getting stuck in a local maximum) at the expense of more computation time.
- **stop_on_first** (`bool`) – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.
- **max_anchor_size** (`Optional[int]`) – Maximum number of features in result.
- **min_samples_start** (`int`) – Min number of initial samples.
- **n_covered_ex** (`int`) – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with *desired_label* are stored).
- **binary_cache_size** (`int`) – The result search pre-allocates *binary_cache_size* batches for storing the binary arrays returned during sampling.
- **cache_margin** (`int`) – When only $\max(\text{cache_margin}, \text{batch_size})$ positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
- **verbose** (`bool`) – Display updates during the anchor search iterations.
- **verbose_every** (`int`) – Frequency of displayed iterations during anchor search process.

Return type

[Explanation](#)

Returns

explanation – *Explanation* object containing the result explaining the instance with additional metadata as attributes. See usage at [AnchorTabular examples](#) for details.

Raises

alibi.exceptions.NotFittedError – If *fit* has not been called prior to calling *explain*.

fit(*train_data*, *disc_perc*=(25, 50, 75), ***kwargs*)

Fit discretizer to train data to bin numerical features into ordered bins and compute statistics for numerical features. Create a mapping between the bin numbers of each discretised numerical feature and the row id in the training set where it occurs.

Parameters

- **train_data** (ndarray) – Representative sample from the training data.
- **disc_perc** (Tuple[Union[int, float], ...]) – List with percentiles (*int*) used for discretization.

Return type

AnchorTabular

instance_label: int

The label of the instance to be explained.

property predictor: Callable | None

Return type

Optional[Callable]

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (Callable) – New predictor function.

Return type

None

```
class alibi.explainers.AnchorText(predictor, sampling_strategy='unknown', nlp=None,
                                  language_model=None, seed=0, **kwargs)
```

Bases: *Explainer*

```
CLASS_SAMPLER = {'language_model': <class
'alibi.explainers.anchors.language_model_text_sampler.LanguageModelSampler'>,
'similarity': <class 'alibi.explainers.anchors.text_samplers.SimilaritySampler'>,
'unknown': <class 'alibi.explainers.anchors.text_samplers.UnknownSampler'>}
```

```
DEFAULTS: Dict[str, Dict] = {'language_model': {'batch_size_lm': 32, 'filling':
'parallel', 'frac_mask_templates': 0.1, 'punctuation':
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~', 'sample_proba': 0.5, 'sample_punctuation':
False, 'stopwords': [], 'temperature': 1.0, 'top_n': 100, 'use_proba': False},
'similarity': {'sample_proba': 0.5, 'temperature': 1.0, 'top_n': 100, 'use_proba':
False}, 'unknown': {'sample_proba': 0.5}}
```

```
SAMPLING_LANGUAGE_MODEL = 'language_model'
```

Language model sampling strategy.

```
SAMPLING_SIMILARITY = 'similarity'
```

Similarity sampling strategy.

SAMPLING_UNKNOWN = 'unknown'

Unknown sampling strategy.

__init__(*predictor*, *sampling_strategy*='unknown', *nlp*=None, *language_model*=None, *seed*=0, ***kwargs*)

Initialize anchor text explainer.

Parameters

- **predictor** ([Callable](#)[\[\[List\[str\]\]](#), ndarray]) – A callable that takes a list of text strings representing N data points as inputs and returns N outputs.
- **sampling_strategy** ([str](#)) – Perturbation distribution method:
 - 'unknown' - replaces words with UNKs.
 - 'similarity' - samples according to a similarity score with the corpus embeddings.
 - 'language_model' - samples according the language model's output distributions.
- **nlp** ([Optional](#)[\[Language\]](#)) – *spaCy* object when sampling method is 'unknown' or 'similarity'.
- **language_model** ([Optional](#)[\[LanguageModel\]](#)) – Transformers masked language model. This is a model that it adheres to the *LanguageModel* interface we define in [alibi.utils.lang_model.LanguageModel](#).
- **seed** ([int](#)) – If set, ensure identical random streams.
- **kwargs** ([Any](#)) – Sampling arguments can be passed as *kwargs* depending on the *sampling_strategy*. Check default arguments defined in:
 - `alibi.explainers.anchor_text.DEFAULT_SAMPLING_UNKNOWN`
 - `alibi.explainers.anchor_text.DEFAULT_SAMPLING_SIMILARITY`
 - `alibi.explainers.anchor_text.DEFAULT_SAMPLING_LANGUAGE_MODEL`

Raises

- [alibi.exceptions.PredictorCallError](#) – If calling *predictor* fails at runtime.
- [alibi.exceptions.PredictorReturnTypeError](#) – If the return type of *predictor* is not *np.ndarray*.

compare_labels(*samples*)

Compute the agreement between a classifier prediction on an instance to be explained and the prediction on a set of samples which have a subset of features fixed to a given value (aka compute the precision of anchors).

Parameters

samples (ndarray) – Samples whose labels are to be compared with the instance label.

Return type

ndarray

Returns

A *numpy* boolean array indicating whether the prediction was the same as the instance label.

explain(*text*, *threshold*=0.95, *delta*=0.1, *tau*=0.15, *batch_size*=100, *coverage_samples*=10000, *beam_size*=1, *stop_on_first*=True, *max_anchor_size*=None, *min_samples_start*=100, *n_covered_ex*=10, *binary_cache_size*=10000, *cache_margin*=1000, *verbose*=False, *verbose_every*=1, ***kwargs*)

Explain instance and return anchor with metadata.

Parameters

- **text** (`str`) – Text instance to be explained.
- **threshold** (`float`) – Minimum anchor precision threshold. The algorithm tries to find an anchor that maximizes the coverage under precision constraint. The precision constraint is formally defined as $P(\text{prec}(A) \geq t) \geq 1 - \delta$, where A is an anchor, t is the *threshold* parameter, δ is the *delta* parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, we are seeking for an anchor having its precision greater or equal than the given *threshold* with a confidence of $(1 - \text{delta})$. A higher value guarantees that the anchors are faithful to the model, but also leads to more computation time. Note that there are cases in which the precision constraint cannot be satisfied due to the quantile-based discretisation of the numerical features. If that is the case, the best (i.e. highest coverage) non-eligible anchor is returned.
- **delta** (`float`) – Significance threshold. $1 - \text{delta}$ represents the confidence threshold for the anchor precision (see *threshold*) and the selection of the best anchor candidate in each iteration (see *tau*).
- **tau** (`float`) – Multi-armed bandit parameter used to select candidate anchors in each iteration. The multi-armed bandit algorithm tries to find within a tolerance *tau* the most promising (i.e. according to the precision) *beam_size* candidate anchor(s) from a list of proposed anchors. Formally, when the *beam_size*=1, the multi-armed bandit algorithm seeks to find an anchor A such that $P(\text{prec}(A) \geq \text{prec}(A^*) - \tau) \geq 1 - \delta$, where A^* is the anchor with the highest true precision (which we don't know), τ is the *tau* parameter, δ is the *delta* parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, in each iteration, the algorithm returns with a probability of at least $1 - \text{delta}$ an anchor A with a precision within an error tolerance of *tau* from the precision of the highest true precision anchor A^* . A bigger value for *tau* means faster convergence but also looser anchor conditions.
- **batch_size** (`int`) – Batch size used for sampling. The Anchor algorithm will query the black-box model in batches of size *batch_size*. A larger *batch_size* gives more confidence in the anchor, again at the expense of computation time since it involves more model prediction calls.
- **coverage_samples** (`int`) – Number of samples used to estimate coverage from during anchor search.
- **beam_size** (`int`) – Number of candidate anchors selected by the multi-armed bandit algorithm in each iteration from a list of proposed anchors. A bigger beam width can lead to a better overall anchor (i.e. prevents the algorithm of getting stuck in a local maximum) at the expense of more computation time.
- **stop_on_first** (`bool`) – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.
- **max_anchor_size** (`Optional[int]`) – Maximum number of features to include in an anchor.
- **min_samples_start** (`int`) – Number of samples used for anchor search initialisation.
- **n_covered_ex** (`int`) – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with predicted label are stored).
- **binary_cache_size** (`int`) – The anchor search pre-allocates *binary_cache_size* batches for storing the boolean arrays returned during sampling.

- **cache_margin** (`int`) – When only `max(cache_margin, batch_size)` positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
- **verbose** (`bool`) – Display updates during the anchor search iterations.
- **verbose_every** (`int`) – Frequency of displayed iterations during anchor search process.
- ****kwargs** (`Any`) – Other keyword arguments passed to the anchor beam search and the text sampling and perturbation functions.

Return type*Explanation***Returns**

Explanation object containing the anchor explaining the instance with additional metadata as attributes. Contains the following data-related attributes –

- *anchor* : `List[str]` - a list of words in the proposed anchor.
- *precision* : `float` - the fraction of times the sampled instances where the anchor holds yields the same prediction as the original instance. The precision will always be threshold for a valid anchor.
- *coverage* : `float` - the fraction of sampled instances the anchor applies to.

model: `spacy.language.Language` | *LanguageModel*

Language model to be used.

perturbation: *Any*

Perturbation method.

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (`Callable`) – New predictor function.

Return type*None*

sampler(*anchor*, *num_samples*, *compute_labels=True*)

Generate perturbed samples while maintaining features in positions specified in anchor unchanged.

Parameters

- **anchor** (`Tuple[int, tuple]`) –
 - `int` - the position of the anchor in the input batch.
 - `tuple` - the anchor itself, a list of words to be kept unchanged.
- **num_samples** (`int`) – Number of generated perturbed samples.
- **compute_labels** (`bool`) – If True, an array of comparisons between predictions on perturbed samples and instance to be explained is returned.

Return type`Union[List[Union[ndarray, float, int]], List[ndarray]]`**Returns**

- If `compute_labels=True`, a list containing the following is returned –

- *covered_true* - perturbed examples where the anchor applies and the model prediction on perturbation is the same as the instance prediction.
- *covered_false* - perturbed examples where the anchor applies and the model prediction is NOT the same as the instance prediction.
- *labels* - num_samples ints indicating whether the prediction on the perturbed sample matches (1) the label of the instance to be explained or not (0).
- *data* - Matrix with 1s and 0s indicating whether a word in the text has been perturbed for each sample.
- *-1.0* - indicates exact coverage is not computed for this algorithm.
- *anchor[0]* - position of anchor in the batch request.
- *Otherwise, a list containing the data matrix only is returned.*

```
class alibi.explainers.CEM(predict, mode, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0,
10000000000.0), gamma=0.0, ae_model=None, learning_rate_init=0.01,
max_iterations=1000, c_init=10.0, c_steps=10, eps=(0.001, 0.001),
clip=(-100.0, 100.0), update_num_grad=1, no_info_val=None, write_dir=None,
sess=None)
```

Bases: [Explainer](#), [FitMixin](#)

```
__init__(predict, mode, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0),
gamma=0.0, ae_model=None, learning_rate_init=0.01, max_iterations=1000, c_init=10.0,
c_steps=10, eps=(0.001, 0.001), clip=(-100.0, 100.0), update_num_grad=1, no_info_val=None,
write_dir=None, sess=None)
```

Initialize contrastive explanation method. Paper: <https://arxiv.org/abs/1802.07623>

Parameters

- **predict** ([Union](#)[[Callable](#)[[[ndarray](#)], [ndarray](#)], [Model](#)]) – *tensorflow* model or any other model's prediction function returning class probabilities.
- **mode** ([str](#)) – Find pertinent negatives (PN) or pertinent positives (PP).
- **shape** ([tuple](#)) – Shape of input data starting with batch size.
- **kappa** ([float](#)) – Confidence parameter for the attack loss term.
- **beta** ([float](#)) – Regularization constant for L1 loss term.
- **feature_range** ([tuple](#)) – Tuple with min and max ranges to allow for perturbed instances. Min and max ranges can be *float* or *numpy* arrays with dimension (1x nb of features) for feature-wise ranges.
- **gamma** ([float](#)) – Regularization constant for optional auto-encoder loss term.
- **ae_model** ([Optional](#)[[Model](#)]) – Optional auto-encoder model used for loss regularization.
- **learning_rate_init** ([float](#)) – Initial learning rate of optimizer.
- **max_iterations** ([int](#)) – Maximum number of iterations for finding a PN or PP.
- **c_init** ([float](#)) – Initial value to scale the attack loss term.
- **c_steps** ([int](#)) – Number of iterations to adjust the constant scaling the attack loss term.
- **eps** ([tuple](#)) – If numerical gradients are used to compute $dL/dx = (dL/dp) * (dp/dx)$, then *eps[0]* is used to calculate dL/dp and *eps[1]* is used for dp/dx . *eps[0]* and *eps[1]* can be a

combination of *float* values and *numpy* arrays. For *eps[0]*, the array dimension should be (1x nb of prediction categories) and for *eps[1]* it should be (1x nb of features).

- **clip** (*tuple*) – Tuple with *min* and *max* clip ranges for both the numerical gradients and the gradients obtained from the *tensorflow* graph.
- **update_num_grad** (*int*) – If numerical gradients are used, they will be updated every *update_num_grad* iterations.
- **no_info_val** (*Union[[float](#), ndarray, [None](#)]*) – Global or feature-wise value considered as containing no information.
- **write_dir** (*Optional[str]*) – Directory to write *tensorboard* files to.
- **sess** (*Optional[Session]*) – Optional *tensorflow* session that will be used if passed instead of creating or inferring one internally.

attack(*X, Y, verbose=False*)

Find pertinent negative or pertinent positive for instance *X* using a fast iterative shrinkage-thresholding algorithm (FISTA).

Parameters

- **X** (*ndarray*) – Instance to attack.
- **Y** (*ndarray*) – Labels for *X*.
- **verbose** (*bool*) – Print intermediate results of optimization if *True*.

Return type

Tuple[*ndarray*, *Tuple*[*ndarray*, *ndarray*]]

Returns

Overall best attack and gradients for that attack.

explain(*X, Y=None, verbose=False*)

Explain instance and return PP or PN with metadata.

Parameters

- **X** (*ndarray*) – Instances to attack.
- **Y** (*Optional*[*ndarray*]) – Labels for *X*.
- **verbose** (*bool*) – Print intermediate results of optimization if *True*.

Return type

Explanation

Returns

explanation – *Explanation* object containing the PP or PN with additional metadata as attributes. See usage at [CEM examples](#) for details.

fit(*train_data, no_info_type='median'*)

Get ‘no information’ values from the training data.

Parameters

- **train_data** (*ndarray*) – Representative sample from the training data.
- **no_info_type** (*str*) – Median or mean value by feature supported.

Return type

CEM

get_gradients(*X*, *Y*)

Compute numerical gradients of the attack loss term: $dL/dx = (dL/dP)*(dP/dx)$ with $L = loss_attack_s$; $P = predict$; $x = adv_s$

Parameters

- **X** (ndarray) – Instance around which gradient is evaluated.
- **Y** (ndarray) – One-hot representation of instance labels.

Return type

ndarray

Returns

Array with gradients.

loss_fn(*pred_proba*, *Y*)

Compute the attack loss.

Parameters

- **pred_proba** (ndarray) – Prediction probabilities of an instance.
- **Y** (ndarray) – One-hot representation of instance labels.

Return type

ndarray

Returns

Loss of the attack.

perturb(*X*, *eps*, *proba=False*)

Apply perturbation to instance or prediction probabilities. Used for numerical calculation of gradients.

Parameters

- **X** (ndarray) – Array to be perturbed.
- **eps** (`Union[float, ndarray]`) – Size of perturbation.
- **proba** (`bool`) – If True, the net effect of the perturbation needs to be 0 to keep the sum of the probabilities equal to 1.

Return type

`Tuple[ndarray, ndarray]`

Returns

Instances where a positive and negative perturbation is applied.

reset_predictor(*predictor*)

Resets the predictor function/model.

Parameters

predictor (`Union[Callable, Model]`) – New predictor function/model.

Return type

`None`

```
class alibi.explainers.Counterfactual(predict_fn, shape, distance_fn='l1', target_proba=1.0,
                                     target_class='other', max_iter=1000, early_stop=50,
                                     lam_init=0.1, max_lam_steps=10, tol=0.05,
                                     learning_rate_init=0.1, feature_range=(-10000000000.0,
                                     10000000000.0), eps=0.01, init='identity', decay=True,
                                     write_dir=None, debug=False, sess=None)
```

Bases: [Explainer](#)

```
__init__(predict_fn, shape, distance_fn='l1', target_proba=1.0, target_class='other', max_iter=1000,
          early_stop=50, lam_init=0.1, max_lam_steps=10, tol=0.05, learning_rate_init=0.1,
          feature_range=(-10000000000.0, 10000000000.0), eps=0.01, init='identity', decay=True,
          write_dir=None, debug=False, sess=None)
```

Initialize counterfactual explanation method based on Wachter et al. (2017)

Parameters

- **predict_fn** ([Union](#)[[Callable](#)[[[ndarray](#)], [ndarray](#)], [Model](#)]) – *tensorflow* model or any other model's prediction function returning class probabilities.
- **shape** ([Tuple](#)[[int](#), ...]) – Shape of input data starting with batch size.
- **distance_fn** ([str](#)) – Distance function to use in the loss term.
- **target_proba** ([float](#)) – Target probability for the counterfactual to reach.
- **target_class** ([Union](#)[[str](#), [int](#)]) – Target class for the counterfactual to reach, one of 'other', 'same' or an integer denoting desired class membership for the counterfactual instance.
- **max_iter** ([int](#)) – Maximum number of iterations to run the gradient descent for (inner loop).
- **early_stop** ([int](#)) – Number of steps after which to terminate gradient descent if all or none of found instances are solutions.
- **lam_init** ([float](#)) – Initial regularization constant for the prediction part of the Wachter loss.
- **max_lam_steps** ([int](#)) – Maximum number of times to adjust the regularization constant (outer loop) before terminating the search.
- **tol** ([float](#)) – Tolerance for the counterfactual target probability.
- **learning_rate_init** – Initial learning rate for each outer loop of *lambda*.
- **feature_range** ([Union](#)[[Tuple](#), [str](#)]) – Tuple with *min* and *max* ranges to allow for perturbed instances. *Min* and *max* ranges can be *float* or *numpy* arrays with dimension (1 x nb of features) for feature-wise ranges.
- **eps** ([Union](#)[[float](#), [ndarray](#)]) – Gradient step sizes used in calculating numerical gradients, defaults to a single value for all features, but can be passed an array for feature-wise step sizes.
- **init** ([str](#)) – Initialization method for the search of counterfactuals, currently must be 'identity'.
- **decay** ([bool](#)) – Flag to decay learning rate to zero for each outer loop over *lambda*.
- **write_dir** ([Optional](#)[[str](#)]) – Directory to write *tensorboard* files to.
- **debug** ([bool](#)) – Flag to write *tensorboard* summaries for debugging.
- **sess** ([Optional](#)[[Session](#)]) – Optional *tensorflow* session that will be used if passed instead of creating or inferring one internally.

explain(X)

Explain an instance and return the counterfactual with metadata.

Parameters

X (ndarray) – Instance to be explained.

Return type

Explanation

Returns

explanation – *Explanation* object containing the counterfactual with additional metadata as attributes. See usage at [Counterfactual examples](#) for details.

fit(X, y)

Fit method - currently unused as the counterfactual search is fully unsupervised.

Parameters

- **X** (ndarray) – Not used. Included for consistency.
- **y** (Optional[ndarray]) – Not used. Included for consistency.

Return type

Counterfactual

Returns

self – Explainer itself.

reset_predictor(predictor)

Resets the predictor function/model.

Parameters

predictor (Union[Callable, Model]) – New predictor function/model.

Return type

None

```
class alibi.explainers.CounterfactualProto(predict, shape, kappa=0.0, beta=0.1,
                                           feature_range=(-10000000000.0, 10000000000.0),
                                           gamma=0.0, ae_model=None, enc_model=None, theta=0.0,
                                           cat_vars=None, ohe=False, use_kdtree=False,
                                           learning_rate_init=0.01, max_iterations=1000, c_init=10.0,
                                           c_steps=10, eps=(0.001, 0.001), clip=(-1000.0, 1000.0),
                                           update_num_grad=1, write_dir=None, sess=None)
```

Bases: [Explainer](#), [FitMixin](#)

```
__init__(predict, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0),
          gamma=0.0, ae_model=None, enc_model=None, theta=0.0, cat_vars=None, ohe=False,
          use_kdtree=False, learning_rate_init=0.01, max_iterations=1000, c_init=10.0, c_steps=10,
          eps=(0.001, 0.001), clip=(-1000.0, 1000.0), update_num_grad=1, write_dir=None, sess=None)
```

Initialize prototypical counterfactual method.

Parameters

- **predict** (Union[Callable[[ndarray], ndarray], Model]) – *tensorflow* model or any other model's prediction function returning class probabilities.
- **shape** (tuple) – Shape of input data starting with batch size.
- **kappa** (float) – Confidence parameter for the attack loss term.
- **beta** (float) – Regularization constant for L1 loss term.

- **feature_range** (`Tuple[Union[float, ndarray], Union[float, ndarray]]`) – Tuple with *min* and *max* ranges to allow for perturbed instances. *Min* and *max* ranges can be *float* or *numpy* arrays with dimension (1x nb of features) for feature-wise ranges.
- **gamma** (`float`) – Regularization constant for optional auto-encoder loss term.
- **ae_model** (`Optional[Model]`) – Optional auto-encoder model used for loss regularization.
- **enc_model** (`Optional[Model]`) – Optional encoder model used to guide instance perturbations towards a class prototype.
- **theta** (`float`) – Constant for the prototype search loss term.
- **cat_vars** (`Optional[Dict[int, int]]`) – Dict with as keys the categorical columns and as values the number of categories per categorical variable.
- **ohe** (`bool`) – Whether the categorical variables are one-hot encoded (OHE) or not. If not OHE, they are assumed to have ordinal encodings.
- **use_kdtree** (`bool`) – Whether to use k-d trees for the prototype loss term if no encoder is available.
- **learning_rate_init** (`float`) – Initial learning rate of optimizer.
- **max_iterations** (`int`) – Maximum number of iterations for finding a counterfactual.
- **c_init** (`float`) – Initial value to scale the attack loss term.
- **c_steps** (`int`) – Number of iterations to adjust the constant scaling the attack loss term.
- **eps** (`tuple`) – If numerical gradients are used to compute $dL/dx = (dL/dp) * (dp/dx)$, then *eps*[0] is used to calculate dL/dp and *eps*[1] is used for dp/dx . *eps*[0] and *eps*[1] can be a combination of *float* values and *numpy* arrays. For *eps*[0], the array dimension should be (1x nb of prediction categories) and for *eps*[1] it should be (1x nb of features).
- **clip** (`tuple`) – Tuple with min and max clip ranges for both the numerical gradients and the gradients obtained from the *tensorflow* graph.
- **update_num_grad** (`int`) – If numerical gradients are used, they will be updated every *update_num_grad* iterations.
- **write_dir** (`Optional[str]`) – Directory to write *tensorboard* files to.
- **sess** (`Optional[Session]`) – Optional *tensorflow* session that will be used if passed instead of creating or inferring one internally.

attack(*X*, *Y*, *target_class=None*, *k=None*, *k_type='mean'*, *threshold=0.0*, *verbose=False*, *print_every=100*, *log_every=100*)

Find a counterfactual (CF) for instance *X* using a fast iterative shrinkage-thresholding algorithm (FISTA).

Parameters

- **X** (`ndarray`) – Instance to attack.
- **Y** (`ndarray`) – Labels for *X* as one-hot-encoding.
- **target_class** (`Optional[list]`) – List with target classes used to find closest prototype. If *None*, the nearest prototype except for the predict class on the instance is used.
- **k** (`Optional[int]`) – Number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class if an encoder is used and to 1 for k-d trees.

- **k_type** (`str`) – Use either the average encoding of the *k* nearest instances in a class (`k_type='mean'`) or the *k*-nearest encoding in the class (`k_type='point'`) to define the prototype of that class. Only relevant if an encoder is used to define the prototypes.
- **threshold** (`float`) – Threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements.
- **verbose** (`bool`) – Print intermediate results of optimization if `True`.
- **print_every** (`int`) – Print frequency if `verbose` is `True`.
- **log_every** (`int`) – *tensorboard* log frequency if write directory is specified.

Return type`Tuple[ndarray, Tuple[ndarray, ndarray]]`**Returns***Overall best attack and gradients for that attack.*

explain(*X*, *Y=None*, *target_class=None*, *k=None*, *k_type='mean'*, *threshold=0.0*, *verbose=False*, *print_every=100*, *log_every=100*)

Explain instance and return counterfactual with metadata.

Parameters

- **X** (`ndarray`) – Instances to attack.
- **Y** (`Optional[ndarray]`) – Labels for *X* as one-hot-encoding.
- **target_class** (`Optional[list]`) – List with target classes used to find closest prototype. If `None`, the nearest prototype except for the predict class on the instance is used.
- **k** (`Optional[int]`) – Number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class if an encoder is used and to 1 for *k*-d trees.
- **k_type** (`str`) – Use either the average encoding of the *k* nearest instances in a class (`k_type='mean'`) or the *k*-nearest encoding in the class (`k_type='point'`) to define the prototype of that class. Only relevant if an encoder is used to define the prototypes.
- **threshold** (`float`) – Threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements.
- **verbose** (`bool`) – Print intermediate results of optimization if `True`.
- **print_every** (`int`) – Print frequency if `verbose` is `True`.
- **log_every** (`int`) – *tensorboard* log frequency if write directory is specified

Return type*Explanation***Returns**

explanation – *Explanation* object containing the counterfactual with additional metadata as attributes. See usage at [CFProto examples](#) for details.

fit(*train_data*, *trustscore_kwargs=None*, *d_type='abdm'*, *w=None*, *disc_perc=(25, 50, 75)*, *standardize_cat_vars=False*, *smooth=1.0*, *center=True*, *update_feature_range=True*)

Get prototypes for each class using the encoder or k-d trees. The prototypes are used for the encoder loss term or to calculate the optional trust scores.

Parameters

- **train_data** (ndarray) – Representative sample from the training data.
- **trustscore_kwargs** (Optional[dict]) – Optional arguments to initialize the trust scores method.
- **d_type** (str) – Pairwise distance metric used for categorical variables. Currently, 'abdm', 'mvdm' and 'abdm-mvdm' are supported. 'abdm' infers context from the other variables while 'mvdm' uses the model predictions. 'abdm-mvdm' is a weighted combination of the two metrics.
- **w** (Optional[float]) – Weight on 'abdm' (between 0. and 1.) distance if *d_type* equals 'abdm-mvdm'.
- **disc_perc** (Sequence[Union[int, float]]) – List with percentiles used in binning of numerical features used for the 'abdm' and 'abdm-mvdm' pairwise distance measures.
- **standardize_cat_vars** (bool) – Standardize numerical values of categorical variables if True.
- **smooth** (float) – Smoothing exponent between 0 and 1 for the distances. Lower values will smooth the difference in distance metric between different features.
- **center** (bool) – Whether to center the scaled distance measures. If False, the min distance for each feature except for the feature with the highest raw max distance will be the lower bound of the feature range, but the upper bound will be below the max feature range.
- **update_feature_range** (bool) – Update feature range with scaled values.

Return type

CounterfactualProto

get_gradients(*X*, *Y*, *grads_shape*, *cat_vars_ord*)

Compute numerical gradients of the attack loss term: $dL/dx = (dL/dP)*(dP/dx)$ with $L = loss_attack_s$; $P = predict$; $x = adv_s$.

Parameters

- **X** (ndarray) – Instance around which gradient is evaluated.
- **Y** (ndarray) – One-hot representation of instance labels.
- **grads_shape** (tuple) – Shape of gradients.
- **cat_vars_ord** (dict) – Dict with as keys the categorical columns and as values the number of categories per categorical variable.

Return type

ndarray

Returns

Array with gradients.

loss_fn(*pred_proba*, *Y*)

Compute the attack loss.

Parameters

- **pred_proba** (ndarray) – Prediction probabilities of an instance.

- **Y** (ndarray) – One-hot representation of instance labels.

Return type
ndarray

Returns
Loss of the attack.

reset_predictor(predictor)

Resets the predictor function/model.

Parameters
predictor (Union[Callable, Model]) – New predictor function/model.

Return type
None

score(X, adv_class, orig_class, eps=1e-10)

Parameters

- **X** (ndarray) – Instance to encode and calculate distance metrics for.
- **adv_class** (int) – Predicted class on the perturbed instance.
- **orig_class** (int) – Predicted class on the original instance.
- **eps** (float) – Small number to avoid dividing by 0.

Return type
float

Returns
Ratio between the distance to the prototype of the predicted class for the original instance and the prototype of the predicted class for the perturbed instance.

class alibi.explainers.**CounterfactualRL**(predictor, encoder, decoder, coeff_sparsity, coeff_consistency, latent_dim=None, backend='tensorflow', seed=0, **kwargs)

Bases: [Explainer](#), [FitMixin](#)

Counterfactual Reinforcement Learning.

__init__(predictor, encoder, decoder, coeff_sparsity, coeff_consistency, latent_dim=None, backend='tensorflow', seed=0, **kwargs)

Constructor.

Parameters

- **predictor** (Callable[[ndarray], ndarray]) – A callable that takes a *numpy* array of *N* data points as inputs and returns *N* outputs. For classification task, the second dimension of the output should match the number of classes. Thus, the output can be either a soft label distribution or a hard label distribution (i.e. one-hot encoding) without affecting the performance since *argmax* is applied to the predictor's output.
- **encoder** (Union[Model, Module]) – Pretrained encoder network.
- **decoder** (Union[Model, Module]) – Pretrained decoder network.
- **coeff_sparsity** (float) – Sparsity loss coefficient.
- **coeff_consistency** (float) – Consistency loss coefficient.
- **latent_dim** (Optional[int]) – Auto-encoder latent dimension. Can be omitted if the actor network is user specified.

- **backend** (`str`) – Deep learning backend: `'tensorflow' | 'pytorch'`. Default `'tensorflow'`.
- **seed** (`int`) – Seed for reproducibility. The results are not reproducible for `'tensorflow'` backend.
- ****kwargs** – Used to replace any default parameter from `alibi.explainers.cfml_base.DEFAULT_BASE_PARAMS`.

explain(`X`, `Y_t`, `C=None`, `batch_size=100`)

Explains an input instance

Parameters

- **X** (`ndarray`) – Instances to be explained.
- **Y_t** (`ndarray`) – Counterfactual targets.
- **C** (`Optional[ndarray]`) – Conditional vectors. If `None`, it means that no conditioning was used during training (i.e. the `conditional_func` returns `None`).
- **batch_size** (`int`) – Batch size to be used when generating counterfactuals.

Return type

`Explanation`

Returns

explanation – `Explanation` object containing the counterfactual with additional metadata as attributes. See usage at [CFRL examples](#) for details.

fit(`X`)

Fit the model agnostic counterfactual generator.

Parameters

- **X** (`ndarray`) – Training data array.

Return type

`Explainer`

Returns

self – The explainer itself.

classmethod load(`path`, `predictor`)

Load an explainer from disk.

Parameters

- **path** (`Union[str, PathLike]`) – Path to a directory containing the saved explainer.
- **predictor** (`Any`) – Model or prediction function used to originally initialize the explainer.

Return type

`Explainer`

Returns

An explainer instance.

reset_predictor(`predictor`)

Resets the predictor.

Parameters

- **predictor** (`Any`) – New predictor.

Return type

None

save(*path*)

Save an explainer to disk. Uses the *dill* module.

Parameters

path (`Union[str, PathLike]`) – Path to a directory. A new directory will be created if one does not exist.

Return type

None

```
class alibi.explainers.CounterfactualRLTabular(predictor, encoder, decoder, encoder_preprocessor,
                                              decoder_inv_preprocessor, coeff_sparsity,
                                              coeff_consistency, feature_names, category_map,
                                              immutable_features=None, ranges=None,
                                              weight_num=1.0, weight_cat=1.0, latent_dim=None,
                                              backend='tensorflow', seed=0, **kwargs)
```

Bases: [CounterfactualRL](#)

Counterfactual Reinforcement Learning Tabular.

```
__init__(predictor, encoder, decoder, encoder_preprocessor, decoder_inv_preprocessor, coeff_sparsity,
         coeff_consistency, feature_names, category_map, immutable_features=None, ranges=None,
         weight_num=1.0, weight_cat=1.0, latent_dim=None, backend='tensorflow', seed=0, **kwargs)
```

Constructor.

Parameters

- **predictor** (`Callable[[ndarray], ndarray]`) – A callable that takes a *numpy* array of *N* data points as inputs and returns *N* outputs. For classification task, the second dimension of the output should match the number of classes. Thus, the output can be either a soft label distribution or a hard label distribution (i.e. one-hot encoding) without affecting the performance since *argmax* is applied to the predictor's output.
- **encoder** (`Union[Model, Module]`) – Pretrained heterogeneous encoder network.
- **decoder** (`Union[Model, Module]`) – Pretrained heterogeneous decoder network. The output of the decoder must be a list of tensors.
- **encoder_preprocessor** (`Callable`) – Auto-encoder data pre-processor. Depending on the input format, the pre-processor can normalize numerical attributes, transform label encoding to one-hot encoding etc.
- **decoder_inv_preprocessor** (`Callable`) – Auto-encoder data inverse pre-processor. This is the inverse function of the pre-processor. It can denormalize numerical attributes, transform one-hot encoding to label encoding, feature type casting etc.
- **coeff_sparsity** (`float`) – Sparsity loss coefficient.
- **coeff_consistency** (`float`) – Consistency loss coefficient.
- **feature_names** (`List[str]`) – List of feature names. This should be provided by the dataset.
- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible values for a feature. This should be provided by the dataset.
- **immutable_features** (`Optional[List[str]]`) – List of immutable features.

- **ranges** (`Optional[Dict[str, Tuple[int, int]]]`) – Numerical feature ranges. Note that exist numerical features such as 'Age', which are allowed to increase only. We denote those by 'inc_feat'. Similarly, there exist features allowed to decrease only. We denote them by 'dec_feat'. Finally, there are some free feature, which we denote by 'free_feat'. With the previous notation, we can define `range = {'inc_feat': [0, 1], 'dec_feat': [-1, 0], 'free_feat': [-1, 1]}`. 'free_feat' can be omitted, as any unspecified feature is considered free. Having the ranges of a feature `{'feat': [a_low, a_high]}`, when sampling is performed the numerical value will be clipped between `[a_low * (max_val - min_val), a_high * (max_val - min_val)]`, where `a_low` and `a_high` are the minimum and maximum values the feature 'feat'. This implies that `a_low` and `a_high` are not restricted to `{-1, 0}` and `{0, 1}`, but can be any float number in-between `[-1, 0]` and `[0, 1]`.
- **weight_num** (`float`) – Numerical loss weight.
- **weight_cat** (`float`) – Categorical loss weight.
- **latent_dim** (`Optional[int]`) – Auto-encoder latent dimension. Can be omitted if the actor network is user specified.
- **backend** (`str`) – Deep learning backend: 'tensorflow' | 'pytorch'. Default 'tensorflow'.
- **seed** (`int`) – Seed for reproducibility. The results are not reproducible for 'tensorflow' backend.
- ****kwargs** – Used to replace any default parameter from `alibi.explainers.cfml_base.DEFAULT_BASE_PARAMS`.

explain(*X*, *Y_t*, *C=None*, *batch_size=100*, *diversity=False*, *num_samples=1*, *patience=1000*, *tolerance=0.001*)

Computes counterfactuals for the given instances conditioned on the target and the conditional vector.

Parameters

- **X** (`ndarray`) – Input instances to generate counterfactuals for.
- **Y_t** (`ndarray`) – Target labels.
- **C** (`Optional[List[Dict[str, List[Union[float, str]]]]]`) – List of conditional dictionaries. If `None`, it means that no conditioning was used during training (i.e. the `conditional_func` returns `None`). If conditioning was used during training but no conditioning is desired for the current input, an empty list is expected.
- **diversity** (`bool`) – Whether to generate diverse counterfactual set for the given instance. Only supported for a single input instance.
- **num_samples** (`int`) – Number of diversity samples to be generated. Considered only if `diversity=True`.
- **batch_size** (`int`) – Batch size to use when generating counterfactuals.
- **patience** (`int`) – Maximum number of iterations to perform diversity search stops. If -1, the search stops only if the desired number of samples has been found.
- **tolerance** (`float`) – Tolerance to distinguish two counterfactual instances.

Return type

Explanation

Returns

explanation – *Explanation* object containing the counterfactual with additional metadata as attributes. See usage [CFRL examples](#) for details.

fit(X)

Fit the model agnostic counterfactual generator.

Parameters

X (ndarray) – Training data array.

Return type

[Explainer](#)

Returns

self – The explainer itself.

class `alibi.explainers.DistributedAnchorTabular`(*predictor, feature_names, categorical_names=None, dtype=<class 'numpy.float32'>, ohe=False, seed=None*)

Bases: [AnchorTabular](#)

explain(*X, threshold=0.95, delta=0.1, tau=0.15, batch_size=100, coverage_samples=10000, beam_size=1, stop_on_first=False, max_anchor_size=None, min_samples_start=1, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000, verbose=False, verbose_every=1, **kwargs*)

Explains the prediction made by a classifier on instance X. Sampling is done in parallel over a number of cores specified in `kwargs['ncpu']`.

Parameters

- **X** (ndarray) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **threshold** (float) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **delta** (float) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **tau** (float) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **batch_size** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **coverage_samples** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **beam_size** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **stop_on_first** (bool) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **max_anchor_size** (Optional[int]) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **min_samples_start** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **n_covered_ex** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **binary_cache_size** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.

- **cache_margin** (int) – See [alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain\(\)](#).
- **verbose** (bool) – See [alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain\(\)](#).
- **verbose_every** (int) – See [alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain\(\)](#).
- ****kwargs** (Any) – See [alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain\(\)](#).

Return type[Explanation](#)**Returns**

See [alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain\(\)](#) superclass.

fit(*train_data*, *disc_perc*=(25, 50, 75), ***kwargs*)

Creates a list of handles to parallel processes handles that are used for submitting sampling tasks.

Parameters

- **train_data** (ndarray) – See [alibi.explainers.anchors.anchor_tabular.AnchorTabular.fit\(\)](#) superclass.
- **disc_perc** (tuple) – See [alibi.explainers.anchors.anchor_tabular.AnchorTabular.fit\(\)](#) superclass.
- ****kwargs** – See [alibi.explainers.anchors.anchor_tabular.AnchorTabular.fit\(\)](#) superclass.

Return type[AnchorTabular](#)

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (Callable) – New model prediction function.

Return type[None](#)

```
class alibi.explainers.GradientSimilarity(predictor, loss_fn, sim_fn='grad_dot', task='classification',
                                         precompute_grads=False, backend='tensorflow',
                                         device=None, verbose=False)
```

Bases: [BaseSimilarityExplainer](#)

```
__init__(predictor, loss_fn, sim_fn='grad_dot', task='classification', precompute_grads=False,
          backend='tensorflow', device=None, verbose=False)
```

GradientSimilarity explainer.

The gradient similarity explainer is used to find examples in the training data that the predictor considers similar to test instances the user wants to explain. It uses the gradients of the loss between the model output and the training data labels. These are compared using the similarity function specified by *sim_fn*. The *GradientSimilarity* explainer can be applied to models trained for both classification and regression tasks.

Parameters

- **predictor** ([Union](#)[Model, Module]) – Model to explain.

- **loss_fn** (`Union[Callable[[Tensor, Tensor], Tensor], Callable[[Tensor, Tensor], Tensor]]`) – Loss function used. The gradient of the loss function is used to compute the similarity between the test instances and the training set.
- **sim_fn** (`Literal['grad_dot', 'grad_cos', 'grad_asym_dot']`) – Similarity function to use. The 'grad_dot' similarity function computes the dot product of the gradients, see [alibi.explainers.similarity.metrics.dot\(\)](#). The 'grad_cos' similarity function computes the cosine similarity between the gradients, see [alibi.explainers.similarity.metrics.cos\(\)](#). The 'grad_asym_dot' similarity function is similar to 'grad_dot' but is asymmetric, see [alibi.explainers.similarity.metrics.asym_dot\(\)](#).
- **task** (`Literal['classification', 'regression']`) – Type of task performed by the model. If the task is 'classification', the target value passed to the explain method of the test instance can be specified either directly or left as None, if left None we use the model's maximum prediction. If the task is 'regression', the target value of the test instance must be specified directly.
- **precompute_grads** (`bool`) – Whether to precompute the gradients. If False, gradients are computed on the fly otherwise we precompute them which can be faster when it comes to computing explanations. Note this option may be memory intensive if the model is large.
- **backend** (`Literal['tensorflow', 'pytorch']`) – Backend to use.
- **device** (`Union[int, str, torch.device, None]`) – Device to use. If None, the default device for the backend is used. If using *pytorch* backend see [pytorch device docs](#) for correct options. Note that in the *pytorch* backend case this parameter can be a `torch.device`. If using *tensorflow* backend see [tensorflow docs](#) for correct options.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

Raises

- **ValueError** – If the task is not 'classification' or 'regression'.
- **ValueError** – If the sim_fn is not 'grad_dot', 'grad_cos' or 'grad_asym_dot'.
- **ValueError** – If the backend is not 'tensorflow' or 'pytorch'.
- **TypeError** – If the device is not an int, str, torch.device or None for the torch backend option or if the device is not str or None for the tensorflow backend option.

`explain(X, Y=None)`

Explain the predictor's predictions for a given input.

Computes the similarity score between the inputs and the training set. Returns an explainer object containing the scores, the indices of the training set instances sorted by descending similarity and the most similar and least similar instances of the data set for the input. Note that the input may be a single instance or a batch of instances.

Parameters

- **X** (`Union[ndarray, Tensor, Tensor, Any, List[Any]]`) – X can be a *numpy* array, *tensorflow* tensor, *pytorch* tensor of the same shape as the training data or a list of objects, with or without a leading batch dimension. If the batch dimension is missing it's added.
- **Y** (`Union[ndarray, Tensor, Tensor, None]`) – Y can be a *numpy* array, *tensorflow* tensor or a *pytorch* tensor. In the case of a regression task, the Y argument must be present. If the task is classification then Y defaults to the model prediction.

Return type

[Explanation](#)

Returns

Explanation object containing the ordered similarity scores for the test instance(s) with additional metadata as attributes. Contains the following data-related attributes –

- **scores**: `np.ndarray` - similarity scores for each pair of instances in the training and test set sorted in descending order.
- **ordered_indices**: `np.ndarray` - indices of the paired training and test set instances sorted by the similarity score in descending order.
- **most_similar**: `np.ndarray` - 5 most similar instances in the training set for each test instance. The first element is the most similar instance.
- **least_similar**: `np.ndarray` - 5 least similar instances in the training set for each test instance. The first element is the least similar instance.

Raises

- **ValueError** – If *Y* is `None` and the *task* is 'regression'.
- **ValueError** – If the shape of *X* or *Y* does not match the shape of the training or target data.
- **ValueError** – If the fit method has not been called prior to calling this method.

fit(*X_train*, *Y_train*)

Fit the explainer.

The *GradientSimilarity* explainer requires the model gradients over the training data. In the explain method it compares them to the model gradients for the test instance(s). If `precompute_grads=True` on initialization then the gradients are precomputed here and stored. This will speed up the explain method call but storing the gradients may not be feasible for large models.

Parameters

- **X_train** (`Union[ndarray, List[Any]]`) – Training data.
- **Y_train** (`ndarray`) – Training labels.

Return type

Explainer

Returns

self – Returns self.

```
class alibi.explainers.IntegratedGradients(model, layer=None, target_fn=None,
                                          method='gausslegendre', n_steps=50,
                                          internal_batch_size=100)
```

Bases: *Explainer*

```
__init__(model, layer=None, target_fn=None, method='gausslegendre', n_steps=50,
          internal_batch_size=100)
```

An implementation of the integrated gradients method for *tensorflow* models.

For details of the method see the original paper: <https://arxiv.org/abs/1703.01365>.

Parameters

- **model** (`Model`) – *tensorflow* model.
- **layer** (`Union[Callable[[Model], Layer], Layer, None]`) – A layer or a function having as parameter the model and returning a layer with respect to which the gradients are calculated. If not provided, the gradients are calculated with respect to the input. To guarantee

saving and loading of the explainer, the layer has to be specified as a callable which returns a layer given the model. E.g. `lambda model: model.layers[0].embeddings`.

- **target_fn** (`Optional[Callable]`) – A scalar function that is applied to the predictions of the model. This can be used to specify which scalar output the attributions should be calculated for. This can be particularly useful if the desired output is not known before calling the model (e.g. explaining the *argmax* output for a probabilistic classifier, in this case we could pass `target_fn=partial(np.argmax, axis=1)`).
- **method** (`str`) – Method for the integral approximation. Methods available: "riemann_left", "riemann_right", "riemann_middle", "riemann_trapezoid", "gausslegendre".
- **n_steps** (`int`) – Number of step in the path integral approximation from the baseline to the input instance.
- **internal_batch_size** (`int`) – Batch size for the internal batching.

explain(*X*, *forward_kwargs=None*, *baselines=None*, *target=None*, *attribute_to_layer_inputs=False*)

Calculates the attributions for each input feature or element of layer and returns an Explanation object.

Parameters

- **X** (`Union[ndarray, List[ndarray]]`) – Instance for which integrated gradients attribution are computed.
- **forward_kwargs** (`Optional[dict]`) – Input keyword args. If it's not None, it must be a dict with *numpy* arrays as values. The first dimension of the arrays must correspond to the number of examples. It will be repeated for each of *n_steps* along the integrated path. The attributions are not computed with respect to these arguments.
- **baselines** (`Union[int, float, ndarray, List[int], List[float], List[ndarray], None]`) – Baselines (starting point of the path integral) for each instance. If the passed value is an *np.ndarray* must have the same shape as *X*. If not provided, all features values for the baselines are set to 0.
- **target** (`Union[int, list, ndarray, None]`) – Defines which element of the model output is considered to compute the gradients. Target can be a numpy array, a list or a numeric value. Numeric values are only valid if the model's output is a rank-*n* tensor with $n \leq 2$ (regression and classification models). If a numeric value is passed, the gradients are calculated for the same element of the output for all data points. For regression models whose output is a scalar, target should not be provided. For classification models *target* can be either the true classes or the classes predicted by the model. It must be provided for classification models and regression models whose output is a vector. If the model's output is a rank-*n* tensor with $n > 2$, the target must be a rank-2 numpy array or a list of lists (a matrix) with dimensions *nb_samples* X (*n*-1) .
- **attribute_to_layer_inputs** (`bool`) – In case of layers gradients, controls whether the gradients are computed for the layer's inputs or outputs. If True, gradients are computed for the layer's inputs, if False for the layer's outputs.

Return type

Explanation

Returns

explanation – *Explanation* object including *meta* and *data* attributes with integrated gradients attributions for each feature. See usage at [IG examples](#) for details.

reset_predictor(*predictor*)

Resets the predictor model.

Parameters

predictor (Model) – New prediction model.

Return type

None

```
class alibi.explainers.KernelShap(predictor, link='identity', feature_names=None,
                                  categorical_names=None, task='classification', seed=None,
                                  distributed_opts=None)
```

Bases: [Explainer](#), [FitMixin](#)

```
__init__(predictor, link='identity', feature_names=None, categorical_names=None, task='classification',
          seed=None, distributed_opts=None)
```

A wrapper around the *shap.KernelExplainer* class. It extends the current *shap* library functionality by allowing the user to specify variable groups in order to treat one-hot encoded categorical as one during sampling. The user can also specify whether to aggregate the *shap* values estimate for the encoded levels of categorical variables as an optional argument to *explain*, if grouping arguments are not passed to *fit*.

Parameters

- **predictor** ([Callable](#)[[[ndarray](#)], [ndarray](#)]) – A callable that takes as an input a *samples x features* array and outputs a *samples x n_outputs* model outputs. The *n_outputs* should represent model output in margin space. If the model outputs probabilities, then the link should be set to 'logit' to ensure correct force plots.
- **link** ([str](#)) – Valid values are 'identity' or 'logit'. A generalized linear model link to connect the feature importance values to the model output. Since the feature importance values, ϕ , sum up to the model output, it often makes sense to connect them to the output with a link function where $link(output - expected_value) = sum(\phi)$. Therefore, for a model which outputs probabilities, `link='logit'` makes the feature effects have log-odds (evidence) units and `link='identity'` means that the feature effects have probability units. Please see this [example](#) for an in-depth discussion about the semantics of explaining the model in the probability or margin space.
- **feature_names** ([Union](#)[[List](#)[[str](#)], [Tuple](#)[[str](#)], [None](#)]) – Used to infer group names when categorical data is treated by grouping and *group_names* input to *fit* is not specified, assuming it has the same length as the *groups* argument of *fit* method. It is also used to compute the *names* field, which appears as a key in each of the values of *explanation.data*['raw']['importances'].
- **categorical_names** ([Optional](#)[[Dict](#)[[int](#), [List](#)[[str](#)]]]) – Keys are feature column indices in the *background_data* matrix (see *fit*). Each value contains strings with the names of the categories for the feature. Used to select the method for background data summarisation (if specified, subsampling is performed as opposed to k-means clustering). In the future it may be used for visualisation.
- **task** ([str](#)) – Can have values 'classification' and 'regression'. It is only used to set the contents of *explanation.data*['raw']['prediction']
- **seed** ([Optional](#)[[int](#)]) – Fixes the random number stream, which influences which subsets are sampled during shap value estimation.
- **distributed_opts** ([Optional](#)[[Dict](#)]) – A dictionary that controls the algorithm distributed execution. See [alibi.explainers.shap_wrappers.DISTRIBUTED_OPTS](#) documentation for details.

```
explain(X, summarise_result=False, cat_vars_start_idx=None, cat_vars_enc_dim=None, **kwargs)
```

Explains the instances in the array *X*.

Parameters

- **X** (`Union`[`ndarray`, `DataFrame`, `spmatrix`]) – Instances to be explained.
- **summarise_result** (`bool`) – Specifies whether the shap values corresponding to dimensions of encoded categorical variables should be summed so that a single shap value is returned for each categorical variable. Both the start indices of the categorical variables (`cat_vars_start_idx`) and the encoding dimensions (`cat_vars_enc_dim`) have to be specified
- **cat_vars_start_idx** (`Optional`[`Sequence`[`int`]]) – The start indices of the categorical variables. If specified, `cat_vars_enc_dim` should also be specified.
- **cat_vars_enc_dim** (`Optional`[`Sequence`[`int`]]) – The length of the encoding dimension for each categorical variable. If specified `cat_vars_start_idx` should also be specified.
- ****kwargs** – Keyword arguments specifying explain behaviour. Valid arguments are:
 - *nsamples* - controls the number of predictor calls and therefore runtime.
 - *ll_reg* - the algorithm is exponential in the feature dimension. If set to *auto* the algorithm will first run a feature selection algorithm to select the top features, provided the fraction of sampled sets of missing features is less than 0.2 from the number of total subsets. The Akaike Information Criterion is used in this case. See our examples for more details about available settings for this parameter. Note that by first running a feature selection step, the shapley values of the remainder of the features will be different to those estimated from the entire set.

For more details, please see the shap library [documentation](#) .

Return type

Explanation

Returns

explanation – An explanation object containing the shap values and prediction in the *data* field, along with a *meta* field containing additional data. See usage at [KernelSHAP examples](#) for details.

fit(*background_data*, *summarise_background=False*, *n_background_samples=300*, *group_names=None*, *groups=None*, *weights=None*, ***kwargs*)

This takes a background dataset (usually a subsample of the training set) as an input along with several user specified options and initialises a *KernelShap* explainer. The runtime of the algorithm depends on the number of samples in this dataset and on the number of features in the dataset. To reduce the size of the dataset, the *summarise_background* option and *n_background_samples* should be used. To reduce the feature dimensionality, encoded categorical variables can be treated as one during the feature perturbation process; this decreases the effective feature dimensionality, can reduce the variance of the shap values estimation and reduces slightly the number of calls to the predictor. Further runtime savings can be achieved by changing the *nsamples* parameter in the call to *explain*. Runtime reduction comes with an accuracy trade-off, so it is better to experiment with a runtime reduction method and understand results stability before using the system.

Parameters

- **background_data** (`Union`[`ndarray`, `spmatrix`, `DataFrame`, `Data`]) – Data used to estimate feature contributions and baseline values for force plots. The rows of the background data should represent samples and the columns features.
- **summarise_background** (`Union`[`bool`, `str`]) – A large background dataset impacts the runtime and memory footprint of the algorithm. By setting this argument to `True`, only *n_background_samples* from the provided data are selected. If *group_names* or *groups* arguments are specified, the algorithm assumes that the data contains categorical variables so

the records are selected uniformly at random. Otherwise, *shap.kmeans* (a wrapper around *sklearn* k-means implementation) is used for selection. If set to 'auto', a default of *KERNEL_SHAP_BACKGROUND_THRESHOLD* samples is selected.

- **n_background_samples** (`int`) – The number of samples to keep in the background dataset if `summarise_background=True`.
- **groups** (`Optional[List[Union[Tuple[int], List[int]]]]`) – A list containing sub-lists specifying the indices of features belonging to the same group.
- **group_names** (`Union[List[str], Tuple[str], None]`) – If specified, this array is used to treat groups of features as one during feature perturbation. This feature can be useful, for example, to treat encoded categorical variables as one and can result in computational savings (this may require adjusting the *nsamples* parameter).
- **weights** (`Union[List[float], Tuple[float], ndarray, None]`) – A sequence or array of weights. This is used only if grouping is specified and assigns a weight to each point in the dataset.
- ****kwargs** – Expected keyword arguments include *keep_index* (bool) and should be used if a data frame containing an index column is passed to the algorithm.

Return type

KernelShap

reset_predictor(*predictor*)

Resets the prediction function.

Parameters

predictor (`Callable`) – New prediction function.

Return type

`None`

class `alibi.explainers.PartialDependence`(*predictor*, *feature_names=None*, *categorical_names=None*, *target_names=None*, *verbose=False*)

Bases: *PartialDependenceBase*

Black-box implementation of partial dependence for tabular datasets. Supports multiple feature interactions.

__init__(*predictor*, *feature_names=None*, *categorical_names=None*, *target_names=None*, *verbose=False*)

Initialize black-box model implementation of partial dependence.

Parameters

- **predictor** (`Callable[[ndarray], ndarray]`) – A prediction function which receives as input a *numpy* array of size $N \times F$ and outputs a *numpy* array of size N (i.e. $(N,)$ or $N \times T$, where N is the number of input instances, F is the number of features and T is the number of targets).
- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.
- **categorical_names** (`Optional[Dict[int, List[str]]]`) – Dictionary where keys are feature columns and values are the categories for the feature. Necessary to identify the categorical features in the dataset. An example for *categorical_names* would be:

```
category_map = {0: ["married", "divorced"], 3: ["high school diploma", "master's degree"]}
```

- **target_names** (`Optional[List[str]]`) – A list of target/output names used for displaying results.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

Notes

The length of the *target_names* should match the number of columns returned by a call to the *predictor*. For example, in the case of a binary classifier, if the predictor outputs a decision score (i.e. uses the *decision_function* method) which returns one column, then the length of the *target_names* should be one. On the other hand, if the predictor outputs a prediction probability (i.e. uses the *predict_proba* method) which returns two columns (one for the negative class and one for the positive class), then the length of the *target_names* should be two.

explain(*X*, *features=None*, *kind='average'*, *percentiles=(0.0, 1.0)*, *grid_resolution=100*, *grid_points=None*)

Calculates the partial dependence for each feature and/or tuples of features with respect to the all targets and the reference dataset *X*.

Parameters

- **X** (`ndarray`) – A $N \times F$ tabular dataset used to calculate partial dependence curves. This is typically the training dataset or a representative sample.
- **features** (`Optional[List[Union[int, Tuple[int, int]]]]`) – An optional list of features or tuples of features for which to calculate the partial dependence. If not provided, the partial dependence will be computed for every single features in the dataset. Some example for *features* would be: `[0, 2]`, `[0, 2, (0, 2)]`, `[(0, 2)]`, where 0 and 2 correspond to column 0 and 2 in *X*, respectively.
- **kind** (`Literal['average', 'individual', 'both']`) – If set to 'average', then only the partial dependence (PD) averaged across all samples from the dataset is returned. If set to 'individual', then only the individual conditional expectation (ICE) is returned for each data point from the dataset. Otherwise, if set to 'both', then both the PD and the ICE are returned.
- **percentiles** (`Tuple[float, float]`) – Lower and upper percentiles used to limit the feature values to potentially remove outliers from low-density regions. Note that for features with not many data points with large/low values, the PD estimates are less reliable in those extreme regions. The values must be in `[0, 1]`. Only used with *grid_resolution*.
- **grid_resolution** (`int`) – Number of equidistant points to split the range of each target feature. Only applies if the number of unique values of a target feature in the reference dataset *X* is greater than the *grid_resolution* value. For example, consider a case where a feature can take the following values: `[0.1, 0.3, 0.35, 0.351, 0.4, 0.41, 0.44, ..., 0.5, 0.54, 0.56, 0.6, 0.65, 0.7, 0.9]`, and we are not interested in evaluating the marginal effect at every single point as it can become computationally costly (assume hundreds/thousands of points) without providing any additional information for nearby points (e.g., 0.35 and 351). By setting *grid_resolution*=5, the marginal effect is computed for the values `[0.1, 0.3, 0.5, 0.7, 0.9]` instead, which is less computationally demanding and can provide similar insights regarding the model's behaviour. Note that the extreme values of the grid can be controlled using the *percentiles* argument.
- **grid_points** (`Optional[Dict[int, Union[List, ndarray]]]`) – Custom grid points. Must be a *dict* where the keys are the target features indices and the values are monotonically increasing arrays defining the grid points for a numerical feature, and a subset of categorical feature values for a categorical feature. If the *grid_points* are not specified, then the grid will be constructed based on the unique target feature values available in the

dataset X , or based on the `grid_resolution` and `percentiles` (check `grid_resolution` to see when it applies). For categorical features, the corresponding value in the `grid_points` can be specified either as array of strings or array of integers corresponding the label encodings. Note that the label encoding must match the ordering of the values provided in the `categorical_names`.

Return type

Explanation

Returns

explanation – An *Explanation* object containing the data and the metadata of the calculated partial dependence curves. See usage at [Partial dependence examples](#) for details

```
class alibi.explainers.PartialDependenceVariance(predictor, feature_names=None,
                                                categorical_names=None, target_names=None,
                                                verbose=False)
```

Bases: *Explainer*

Implementation of the partial dependence(PD) variance feature importance and feature interaction for tabular datasets. The method measure the importance feature importance as the variance within the PD function. Similar, the potential feature interaction is measured by computing the variance within the two-way PD function by holding one variable constant and letting the other vary. Supports black-box models and the following *sklearn* tree-based models: *GradientBoostingClassifier*, *GradientBoostingRegressor*, *HistGradientBoostingClassifier*, *HistGradientBoostingRegressor*, *DecisionTreeRegressor*, *RandomForestRegressor*.

For details of the method see the original paper: <https://arxiv.org/abs/1805.04755> .

```
__init__(predictor, feature_names=None, categorical_names=None, target_names=None, verbose=False)
```

Initialize black-box/tree-based model implementation for the partial dependence variance feature importance.

Parameters

- **predictor** (`Union[BaseEstimator, Callable[[ndarray], ndarray]]`) – A *sklearn* estimator or a prediction function which receives as input a *numpy* array of size $N \times F$ and outputs a *numpy* array of size N (i.e. $(N,)$ or $N \times T$, where N is the number of input instances, F is the number of features and T is the number of targets).
- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.
- **categorical_names** (`Optional[Dict[int, List[str]]]`) – Dictionary where keys are feature columns and values are the categories for the feature. Necessary to identify the categorical features in the dataset. An example for `categorical_names` would be:

```
category_map = {0: ["married", "divorced"], 3: ["high school diploma", "master's degree"]}
```

- **target_names** (`Optional[List[str]]`) – A list of target/output names used for displaying results.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

Notes

The length of the *target_names* should match the number of columns returned by a call to the *predictor*. For example, in the case of a binary classifier, if the predictor outputs a decision score (i.e. uses the *decision_function* method) which returns one column, then the length of the *target_names* should be one. On the other hand, if the predictor outputs a prediction probability (i.e. uses the *predict_proba* method) which returns two columns (one for the negative class and one for the positive class), then the length of the *target_names* should be two.

```
explain(X, features=None, method='importance', percentiles=(0.0, 1.0), grid_resolution=100,
        grid_points=None)
```

Calculates the variance partial dependence feature importance for each feature with respect to the all targets and the reference dataset *X*.

Parameters

- **X** (ndarray) – A $N \times F$ tabular dataset used to calculate partial dependence curves. This is typically the training dataset or a representative sample.
- **features** (Union[List[int], List[Tuple[int, int]], None]) – A list of features for which to compute the feature importance or a list of feature pairs for which to compute the feature interaction. Some example of *features* would be: [0, 1, 3], [(0, 1), (0, 3), (1, 3)], where 0, 1, and 3 correspond to the columns 0, 1, and 3 in *X*. If not provided, the feature importance or the feature interaction will be computed for every feature or for every combination of feature pairs, depending on the parameter *method*.
- **method** (Literal['importance', 'interaction']) – Flag to specify whether to compute the feature importance or the feature interaction of the elements provided in *features*. Supported values: 'importance' | 'interaction'.
- **percentiles** (Tuple[float, float]) – Lower and upper percentiles used to limit the feature values to potentially remove outliers from low-density regions. Note that for features with not many data points with large/low values, the PD estimates are less reliable in those extreme regions. The values must be in [0, 1]. Only used with *grid_resolution*.
- **grid_resolution** (int) – Number of equidistant points to split the range of each target feature. Only applies if the number of unique values of a target feature in the reference dataset *X* is greater than the *grid_resolution* value. For example, consider a case where a feature can take the following values: [0.1, 0.3, 0.35, 0.351, 0.4, 0.41, 0.44, ..., 0.5, 0.54, 0.56, 0.6, 0.65, 0.7, 0.9], and we are not interested in evaluating the marginal effect at every single point as it can become computationally costly (assume hundreds/thousands of points) without providing any additional information for nearby points (e.g., 0.35 and 351). By setting *grid_resolution*=5, the marginal effect is computed for the values [0.1, 0.3, 0.5, 0.7, 0.9] instead, which is less computationally demanding and can provide similar insights regarding the model's behaviour. Note that the extreme values of the grid can be controlled using the *percentiles* argument.
- **grid_points** (Optional[Dict[int, Union[List, ndarray]]]) – Custom grid points. Must be a *dict* where the keys are the target features indices and the values are monotonically increasing arrays defining the grid points for a numerical feature, and a subset of categorical feature values for a categorical feature. If the *grid_points* are not specified, then the grid will be constructed based on the unique target feature values available in the dataset *X*, or based on the *grid_resolution* and *percentiles* (check *grid_resolution* to see when it applies). For categorical features, the corresponding value in the *grid_points* can be specified either as array of strings or array of integers corresponding the label encodings. Note that the label encoding must match the ordering of the values provided in the *categorical_names*.

Return type*Explanation***Returns**

explanation – An *Explanation* object containing the data and the metadata of the calculated partial dependence curves and feature importance/interaction. See usage at [Partial dependence variance examples](#) for details

```
class alibi.explainers.PermutationImportance(predictor, loss_fns=None, score_fns=None,
                                             feature_names=None, verbose=False)
```

Bases: [Explainer](#)

Implementation of the permutation feature importance for tabular datasets. The method measure the importance of a feature as the relative increase/decrease in the loss/score function when the feature values are permuted. Supports black-box models.

For details of the method see the papers:

- <https://link.springer.com/article/10.1023/A:1010933404324>
- <https://arxiv.org/abs/1801.01489>

```
__init__(predictor, loss_fns=None, score_fns=None, feature_names=None, verbose=False)
```

Initialize the permutation feature importance.

Parameters

- **predictor** ([Callable](#)[[[ndarray](#)], [ndarray](#)]) – A prediction function which receives as input a *numpy* array of size $N \times F$, and outputs a *numpy* array of size N (i.e. $(N,)$) or $N \times T$, where N is the number of input instances, F is the number of features, and T is the number of targets. Note that the output shape must be compatible with the loss and score functions provided in *loss_fns* and *score_fns*.
- **loss_fns** ([Union](#)[[Literal](#)['mean_absolute_error', 'mean_squared_error', 'mean_squared_log_error', 'mean_absolute_percentage_error', 'log_loss'], [List](#)[[Literal](#)['mean_absolute_error', 'mean_squared_error', 'mean_squared_log_error', 'mean_absolute_percentage_error', 'log_loss']], [Callable](#)[[[ndarray](#), [ndarray](#), [Optional](#)[[ndarray](#)]], [float](#)], [Dict](#)[[str](#), [Callable](#)[[[ndarray](#), [ndarray](#), [Optional](#)[[ndarray](#)]], [float](#)]], [None](#)]) – A literal, or a list of literals, or a loss function, or a dictionary of loss functions having as keys the names of the loss functions and as values the loss functions (i.e., lower values are better). The available literal values are described in [alibi.explainers.permutation_importance.LOSS_FNS](#). Note that the *predictor* output must be compatible with every loss function. Every loss function is expected to receive the following arguments:
 - *y_true* : [np.ndarray](#) - a *numpy* array of ground-truth labels.
 - *y_pred* | *y_score* : [np.ndarray](#) - a *numpy* array of model predictions. This corresponds to the output of the model.
 - *sample_weight*: [Optional](#)[[np.ndarray](#)] - a *numpy* array of sample weights.
- **score_fns** ([Union](#)[[Literal](#)['accuracy', 'precision', 'recall', 'f1', 'roc_auc', 'r2'], [List](#)[[Literal](#)['accuracy', 'precision', 'recall', 'f1', 'roc_auc', 'r2']], [Callable](#)[[[ndarray](#), [ndarray](#), [Optional](#)[[ndarray](#)]], [float](#)], [Dict](#)[[str](#), [Callable](#)[[[ndarray](#), [ndarray](#), [Optional](#)[[ndarray](#)]], [float](#)]], [None](#)]) – A literal, or a list or literals, or a score function, or a dictionary of score functions having as keys the names of the score functions and as values the score functions (i.e, higher

values are better). The available literal values are described in [alibi.explainers.permutation_importance.SCORE_FNS](#). As with the *loss_fns*, the *predictor* output must be compatible with every score function and the score function must have the same signature presented in the *loss_fns* parameter description.

- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

explain(*X*, *y*, *features*=None, *method*='estimate', *kind*='ratio', *n_repeats*=50, *sample_weight*=None)

Computes the permutation feature importance for each feature with respect to the given loss or score functions and the dataset (*X*, *y*).

Parameters

- **X** (`ndarray`) – A $N \times F$ input feature dataset used to calculate the permutation feature importance. This is typically the test dataset.
- **y** (`ndarray`) – Ground-truth labels array of size N (i.e. $(N,)$) corresponding the input feature *X*.
- **features** (`Optional[List[Union[int, Tuple[int, ...]]]`) – An optional list of features or tuples of features for which to compute the permutation feature importance. If not provided, the permutation feature importance will be computed for every single features in the dataset. Some example of *features* would be: `[0, 2]`, `[0, 2, (0, 2)]`, `[(0, 2)]`, where 0 and 2 correspond to column 0 and 2 in *X*, respectively.
- **method** (`Literal['estimate', 'exact']`) – The method to be used to compute the feature importance. If set to 'exact', a “switch” operation is performed across all observed pairs, by excluding pairings that are actually observed in the original dataset. This operation is quadratic in the number of samples ($N \times (N - 1)$ samples) and thus can be computationally intensive. If set to 'estimate', the dataset will be divided in half. The values of the first half containing the ground-truth labels the rest of the features (i.e. features that are left intact) is matched with the values of the second half of the permuted features, and the other way around. This method is computationally lighter and provides estimate error bars given by the standard deviation. Note that for some specific loss and score functions, the estimate does not converge to the exact metric value.
- **kind** (`Literal['ratio', 'difference']`) – Whether to report the importance as the loss/score ratio or the loss/score difference. Available values are: 'ratio' | 'difference'.
- **n_repeats** (`int`) – Number of times to permute the feature values. Considered only when *method*='estimate'.
- **sample_weight** (`Optional[ndarray]`) – Optional weight for each sample instance.

Return type

[Explanation](#)

Returns

explanation – An *Explanation* object containing the data and the metadata of the permutation feature importance. See usage at [Permutation feature importance examples](#) for details

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (`Callable`) – New predictor function.

Return type`None`

```
class alibi.explainers.TreePartialDependence(predictor, feature_names=None,
                                             categorical_names=None, target_names=None,
                                             verbose=False)
```

Bases: [`PartialDependenceBase`](#)

Tree-based model *sklearn* implementation of the partial dependence for tabular datasets. Supports multiple feature interactions. This method is faster than the general black-box implementation but is only supported by some tree-based estimators. The computation is based on a weighted tree traversal. For more details on the computation, check the [sklearn documentation page](#). The supported *sklearn* models are: *GradientBoostingClassifier*, *GradientBoostingRegressor*, *HistGradientBoostingClassifier*, *HistGradientBoostingRegressor*, *HistGradientBoostingRegressor*, *DecisionTreeRegressor*, *RandomForestRegressor*.

```
__init__(predictor, feature_names=None, categorical_names=None, target_names=None, verbose=False)
```

Initialize tree-based model *sklearn* implementation of partial dependence.

Parameters

- **predictor** (`BaseEstimator`) – A tree-based *sklearn* estimator.
- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.
- **categorical_names** (`Optional[Dict[int, List[str]]]`) – Dictionary where keys are feature columns and values are the categories for the feature. Necessary to identify the categorical features in the dataset. An example for *categorical_names* would be:

```
category_map = {0: ["married", "divorced"], 3: ["high school diploma", "master's degree"]}
```

- **target_names** (`Optional[List[str]]`) – A list of target/output names used for displaying results.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

Notes

The length of the *target_names* should match the number of columns returned by a call to the *predictor.decision_function*. In the case of a binary classifier, the decision score consists of a single column. Thus, the length of the *target_names* should be one.

```
explain(X, features=None, percentiles=(0.0, 1.0), grid_resolution=100, grid_points=None)
```

Calculates the partial dependence for each feature and/or tuples of features with respect to the all targets and the reference dataset *X*.

Parameters

- **X** (`ndarray`) – A $N \times F$ tabular dataset used to calculate partial dependence curves. This is typically the training dataset or a representative sample.
- **features** (`Optional[List[Union[int, Tuple[int, int]]]]`) – An optional list of features or tuples of features for which to calculate the partial dependence. If not provided, the partial dependence will be computed for every single features in the dataset. Some example for *features* would be: `[0, 2]`, `[0, 2, (0, 2)]`, `[(0, 2)]`, where 0 and 2 correspond to column 0 and 2 in *X*, respectively.

- **percentiles** (`Tuple[float, float]`) – Lower and upper percentiles used to limit the feature values to potentially remove outliers from low-density regions. Note that for features with not many data points with large/low values, the PD estimates are less reliable in those extreme regions. The values must be in [0, 1]. Only used with `grid_resolution`.
- **grid_resolution** (`int`) – Number of equidistant points to split the range of each target feature. Only applies if the number of unique values of a target feature in the reference dataset *X* is greater than the `grid_resolution` value. For example, consider a case where a feature can take the following values: [0.1, 0.3, 0.35, 0.351, 0.4, 0.41, 0.44, ..., 0.5, 0.54, 0.56, 0.6, 0.65, 0.7, 0.9], and we are not interested in evaluating the marginal effect at every single point as it can become computationally costly (assume hundreds/thousands of points) without providing any additional information for nearby points (e.g., 0.35 and 351). By setting `grid_resolution=5`, the marginal effect is computed for the values [0.1, 0.3, 0.5, 0.7, 0.9] instead, which is less computationally demanding and can provide similar insights regarding the model's behaviour. Note that the extreme values of the grid can be controlled using the `percentiles` argument.
- **grid_points** (`Optional[Dict[int, Union[List, ndarray]]]`) – Custom grid points. Must be a *dict* where the keys are the target features indices and the values are monotonically increasing arrays defining the grid points for a numerical feature, and a subset of categorical feature values for a categorical feature. If the `grid_points` are not specified, then the grid will be constructed based on the unique target feature values available in the dataset *X*, or based on the `grid_resolution` and `percentiles` (check `grid_resolution` to see when it applies). For categorical features, the corresponding value in the `grid_points` can be specified either as array of strings or array of integers corresponding the label encodings. Note that the label encoding must match the ordering of the values provided in the `categorical_names`.

Return type

Explanation

```
class alibi.explainers.TreeShap(predictor, model_output='raw', feature_names=None,
                                categorical_names=None, task='classification', seed=None)
```

Bases: *Explainer*, *FitMixin*

```
__init__(predictor, model_output='raw', feature_names=None, categorical_names=None,
          task='classification', seed=None)
```

A wrapper around the `shap.TreeExplainer` class. It adds the following functionality:

1. Input summarisation options to allow control over background dataset size and hence runtime
2. Output summarisation for sklearn models with one-hot encoded categorical variables.

Users are strongly encouraged to familiarise themselves with the algorithm by reading the method overview in the documentation.

Parameters

- **predictor** (*Any*) – A fitted model to be explained. *XGBoost*, *LightGBM*, *CatBoost* and most tree-based *scikit-learn* models are supported. In the future, *Pyspark* could also be supported. Please open an issue if this is a use case for you.
- **model_output** (`str`) – Supported values are: 'raw', 'probability', 'probability_doubled', 'log_loss':
 - 'raw' - the raw model of the output, which varies by task, is explained. This option should always be used if the *fit* is called without arguments. It should also be set to compute shap interaction values. For regression models it is the standard output, for binary classification in *XGBoost* it is the log odds ratio.

- 'probability' - the probability output is explained. This option should only be used if *fit* was called with the *background_data* argument set. The effect of specifying this parameter is that the *shap* library will use this information to transform the shap values computed in margin space (aka using the raw output) to shap values that sum to the probability output by the model plus the model expected output probability. This requires knowledge of the type of output for *predictor* which is inferred by the *shap* library from the model type (e.g., most sklearn models with exception of *sklearn.tree.DecisionTreeClassifier*, *sklearn.ensemble.RandomForestClassifier*, *sklearn.ensemble.ExtraTreesClassifier* output logits) or on the basis of the mapping implemented in the *shap.TreeEnsemble* constructor. Only trees that output log odds and probabilities are supported currently.
- 'probability_doubled' - used for binary classification problem in situations where the model outputs the logits/probabilities for the positive class but shap values for both outcomes are desired. This option should be used only if *fit* was called with the *background_data* argument set. In this case the expected value for the negative class is $1 - \text{expected_value for positive class}$ and the shap values for the negative class are the negative values of the positive class shap values. As before, the explanation happens in the margin space, and the shap values are subsequently adjusted. convert the model output to probabilities. The same considerations as for *probability* apply for this output type too.
- 'log_loss' - logarithmic loss is explained. This option should be used only if *fit* was called with the *background_data* argument set and requires specifying labels, *y*, when calling *explain*. If the objective is squared error, then the transformation $(\text{output} - y)^2$ is applied. For binary cross-entropy objective, the transformation $\log(1 + \exp(\text{output})) - y * \text{output}$ with $y \in \{0, 1\}$. Currently only binary cross-entropy and squared error losses can be explained.
- **feature_names** (`Union[List[str], Tuple[str], None]`) – Used to compute the *names* field, which appears as a key in each of the values of the *importances* sub-field of the response *raw* field.
- **categorical_names** (`Optional[Dict[int, List[str]]]`) – Keys are feature column indices. Each value contains strings with the names of the categories for the feature. Used to select the method for background data summarisation (if specified, subsampling is performed as opposed to kmeans clustering). In the future it may be used for visualisation.
- **task** (`str`) – Can have values 'classification' and 'regression'. It is only used to set the contents of the *prediction* field in the *data['raw']* response field.

Notes

Tree SHAP is an additive attribution method so it is best suited to explaining output in margin space (the entire real line). For discussion related to explaining models in output vs probability space, please consult this [resource](#).

explain(*X*, *y=None*, *interactions=False*, *approximate=False*, *check_additivity=True*, *tree_limit=None*, *summarise_result=False*, *cat_vars_start_idx=None*, *cat_vars_enc_dim=None*, ***kwargs*)

Explains the instances in *X*. *y* should be passed if the model loss function is to be explained, which can be useful in order to understand how various features affect model performance over time. This is only possible if the explainer has been fitted with a background dataset and requires setting *model_output='log_loss'*.

Parameters

- **X** (`Union[ndarray, DataFrame, Pool]`) – Instances to be explained.

- **y** (`Optional[ndarray]`) – Labels corresponding to rows of *X*. Should be passed only if a background dataset was passed to the *fit* method.
- **interactions** (`bool`) – If `True`, the shap value for every feature of every instance in *X* is decomposed into *X.shape[1] - 1* shap value interactions and one main effect. This is only supported if *fit* is called with *background_dataset=None*.
- **approximate** (`bool`) – If `True`, an approximation to the shap values that does not account for feature order is computed. This was proposed by [Ando Sabaas](#) here . Check [this](#) resource for more details. This option is currently only supported for *xgboost* and *sklearn* models.
- **check_additivity** (`bool`) – If `True`, output correctness is ensured if *model_output='raw'* has been passed to the constructor.
- **tree_limit** (`Optional[int]`) – Explain the output of a subset of the first *tree_limit* trees in an ensemble model.
- **summarise_result** (`bool`) – This should be set to `True` only when some of the columns in *X* represent encoded dimensions of a categorical variable and one single shap value per categorical variable is desired. Both *cat_vars_start_idx* and *cat_vars_enc_dim* should be specified as detailed below to allow this.
- **cat_vars_start_idx** (`Optional[Sequence[int]]`) – The start indices of the categorical variables.
- **cat_vars_enc_dim** (`Optional[Sequence[int]]`) – The length of the encoding dimension for each categorical variable.

Return type*Explanation***Returns**

explanation – An *Explanation* object containing the shap values and prediction in the *data* field, along with a *meta* field containing additional data. See usage at [TreeSHAP examples](#) for details.

fit(*background_data=None*, *summarise_background=False*, *n_background_samples=1000*, ***kwargs*)

This function instantiates an explainer which can then be use to explain instances using the *explain* method. If no background dataset is passed, the explainer uses the path-dependent feature perturbation algorithm to explain the values. As such, only the model raw output can be explained and this should be reflected by passing *model_output='raw'* when instantiating the explainer. If a background dataset is passed, the interventional feature perturbation algorithm is used. Using this algorithm, probability outputs can also be explained. Additionally, if the *model_output='log_loss'* option is passed to the explainer constructor, then the model loss function can be explained by passing the labels as the *y* argument to the *explain* method. A limited number of loss functions are supported, as detailed in the constructor documentation.

Parameters

- **background_data** (`Union[ndarray, DataFrame, None]`) – Data used to estimate feature contributions and baseline values for force plots. The rows of the background data should represent samples and the columns features.
- **summarise_background** (`Union[bool, str]`) – A large background dataset may impact the runtime and memory footprint of the algorithm. By setting this argument to `True`, only *n_background_samples* from the provided data are selected. If the *categorical_names* argument has been passed to the constructor, subsampling of the data is used. Otherwise, *shap.kmeans* (a wrapper around *sklearn.kmeans* implementation) is used for selection. If set to `'auto'`, a default of *TREE_SHAP_BACKGROUND_WARNING_THRESHOLD* samples is selected.

- **n_background_samples** (*int*) – The number of samples to keep in the background dataset if `summarise_background=True`.

Return type*TreeShap***reset_predictor**(*predictor*)

Resets the predictor.

Parameters**predictor** (*Any*) – New prediction.**Return type***None*

```
alibi.explainers.plot_ale(exp, features='all', targets='all', n_cols=3, sharey='all', constant=False,
                          ax=None, line_kw=None, fig_kw=None)
```

Plot ALE curves on matplotlib axes.

Parameters

- **exp** – An *Explanation* object produced by a call to the `alibi.explainers.ale.ALE.explain()` method.
- **features** – A list of features for which to plot the ALE curves or 'all' for all features. Can be a mix of integers denoting feature index or strings denoting entries in `exp.feature_names`. Defaults to 'all'.
- **targets** – A list of targets for which to plot the ALE curves or 'all' for all targets. Can be a mix of integers denoting target index or strings denoting entries in `exp.target_names`. Defaults to 'all'.
- **n_cols** – Number of columns to organize the resulting plot into.
- **sharey** – A parameter specifying whether the y-axis of the ALE curves should be on the same scale for several features. Possible values are: 'all' | 'row' | *None*.
- **constant** – A parameter specifying whether the constant zeroth order effects should be added to the ALE first order effects.
- **ax** – A *matplotlib* axes object or a *numpy* array of *matplotlib* axes to plot on.
- **line_kw** – Keyword arguments passed to the `plt.plot` function.
- **fig_kw** – Keyword arguments passed to the `fig.set` function.

ReturnsAn array of *matplotlib* axes with the resulting ALE plots.

```
alibi.explainers.plot_pd(exp, features='all', target=0, n_cols=3, n_ice=100, center=False, pd_limits=None,
                          levels=8, ax=None, sharey='all', pd_num_kw=None, ice_num_kw=None,
                          pd_cat_kw=None, ice_cat_kw=None, pd_num_num_kw=None,
                          pd_num_cat_kw=None, pd_cat_cat_kw=None, fig_kw=None)
```

Plot partial dependence curves on matplotlib axes.

Parameters

- **exp** – An *Explanation* object produced by a call to the `alibi.explainers.partial_dependence.PartialDependence.explain()` method.
- **features** – A list of features entries in the `exp.data['feature_names']` to plot the partial dependence curves for, or 'all' to plot all the explained feature or tuples of features. This includes tuples of features. For example, if `exp.data['feature_names'] = ['temp',`

'hum', ('temp', 'windspeed'))] and we want to plot the partial dependence only for the 'temp' and ('temp', 'windspeed'), then we would set `features=[0, 2]`. Defaults to 'all'.

- **target** – The target name or index for which to plot the partial dependence (PD) curves. Can be a mix of integers denoting target index or strings denoting entries in `exp.meta['params']['target_names']`.
- **n_cols** – Number of columns to organize the resulting plot into.
- **n_ice** – Number of ICE plots to be displayed. Can be
 - a string taking the value 'all' to display the ICE curves for every instance in the reference dataset.
 - an integer for which `n_ice` instances from the reference dataset will be sampled uniformly at random to display their ICE curves.
 - a list of integers, where each integer represents an index of an instance in the reference dataset to display their ICE curves.
- **center** – Boolean flag to center the individual conditional expectation (ICE) curves. As mentioned in [Goldstein et al. \(2014\)](#), the heterogeneity in the model can be difficult to discern when the intercepts of the ICE curves cover a wide range. Centering the ICE curves removes the level effects and helps to visualise the heterogeneous effect.
- **pd_limits** – Minimum and maximum y-limits for all the one-way PD plots. If None will be automatically inferred.
- **levels** – Number of levels in the contour plot.
- **ax** – A `matplotlib` axes object or a `numpy` array of `matplotlib` axes to plot on.
- **sharey** – A parameter specifying whether the y-axis of the PD and ICE curves should be on the same scale for several features. Possible values are: 'all' | 'row' | None.
- **pd_num_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the PD for a numerical feature.
- **ice_num_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the ICE for a numerical feature.
- **pd_cat_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the PD for a categorical feature.
- **ice_cat_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the ICE for a categorical feature.
- **pd_num_num_kw** – Keyword arguments passed to the `matplotlib.pyplot.contourf` function when plotting the PD for two numerical features.
- **pd_num_cat_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the PD for a numerical and a categorical feature.
- **pd_cat_cat_kw** – Keyword arguments passed to the `alibi.utils.visualization.heatmap()` function when plotting the PD for two categorical features.
- **fig_kw** – Keyword arguments passed to the `matplotlib.figure.set` function.

Returns

An array of `plt.Axes` with the resulting partial dependence plots.

```
alibi.explainers.plot_pd_variance(exp, features='all', targets='all', summarise=True, n_cols=3, sort=True,
                                top_k=None, plot_limits=None, ax=None, sharey='all', bar_kw=None,
                                line_kw=None, fig_kw=None)
```

Plot feature importance and feature interaction based on partial dependence curves on *matplotlib* axes.

Parameters

- **exp** (*Explanation*) – An *Explanation* object produced by a call to the *alibi.explainers.pd_variance.PartialDependenceVariance.explain()* method.
- **features** (*Union[List[int], Literal['all']]*) – A list of features entries provided in *feature_names* argument to the *alibi.explainers.pd_variance.PartialDependenceVariance.explain()* method, or 'all' to plot all the explained features. For example, if *feature_names* = ['temp', 'hum', 'windspeed'] and we want to plot the values only for the 'temp' and 'windspeed', then we would set *features*=[0, 2]. Defaults to 'all'.
- **targets** (*Union[List[Union[int, str]], Literal['all']]*) – A target name/index, or a list of target names/indices, for which to plot the feature importance/interaction, or 'all'. Can be a mix of integers denoting target index or strings denoting entries in *exp.meta['params']['target_names']*. By default 'all' to plot the importance for all features or to plot all the feature interactions.
- **summarise** (*bool*) – Whether to plot only the summary of the feature importance/interaction as a bar plot, or plot comprehensive exposition including partial dependence plots and conditional importance plots.
- **n_cols** (*int*) – Number of columns to organize the resulting plot into.
- **sort** (*bool*) – Boolean flag whether to sort the values in descending order.
- **top_k** (*Optional[int]*) – Number of top k values to be displayed if the *sort*=True. If not provided, then all values will be displayed.
- **plot_limits** (*Optional[Tuple[float, float]]*) – Minimum and maximum y-limits for all the line plots. If None will be automatically inferred.
- **ax** (*Union[Axes, ndarray, None]*) – A *matplotlib* axes object or a *numpy* array of *matplotlib* axes to plot on.
- **sharey** (*Optional[Literall['all', 'row']]*) – A parameter specifying whether the y-axis of the PD and ICE curves should be on the same scale for several features. Possible values are: 'all' | 'row' | None.
- **bar_kw** (*Optional[dict]*) – Keyword arguments passed to the *matplotlib.pyplot.barh* function.
- **line_kw** (*Optional[dict]*) – Keyword arguments passed to the *matplotlib.pyplot.plot* function.
- **fig_kw** (*Optional[dict]*) – Keyword arguments passed to the *matplotlib.figure.set* function.

Returns

plt.Axes with the summary/detailed exposition plot of the feature importance or feature interaction.

```
alibi.explainers.plot_permutation_importance(exp, features='all', metric_names='all', n_cols=3,
                                             sort=True, top_k=None, ax=None, bar_kw=None,
                                             fig_kw=None)
```

Plot permutation feature importance on *matplotlib* axes.

Parameters

- **exp** – An *Explanation* object produced by a call to the `alibi.explainers.permutation_importance.PermutationImportance.explain()` method.
- **features** – A list of feature entries provided in `feature_names` argument to the `alibi.explainers.permutation_importance.PermutationImportance.explain()` method, or 'all' to plot all the explained features. For example, consider that the `feature_names = ['temp', 'hum', 'windspeed', 'season']`. If we set `features=None` in the `explain` method, meaning that all the feature were explained, and we want to plot only the values for the 'temp' and 'windspeed', then we would set `features=[0, 2]`. Otherwise, if we set `features=[1, 2, 3]` in the `explain` method, meaning that we explained ['hum', 'windspeed', 'season'], and we want to plot the values only for ['windspeed', 'season'], then we would set `features=[1, 2]` (i.e., their index in the `features` list passed to the `explain` method). Defaults to 'all'.
- **metric_names** – A list of metric entries in the `exp.data['metrics']` to plot the permutation feature importance for, or 'all' to plot the permutation feature importance for all metrics (i.e., loss and score functions). The ordering is given by the concatenation of the loss metrics followed by the score metrics.
- **n_cols** – Number of columns to organize the resulting plot into.
- **sort** – Boolean flag whether to sort the values in descending order.
- **top_k** – Number of top k values to be displayed if the `sort=True`. If not provided, then all values will be displayed.
- **ax** – A *matplotlib* axes object or a *numpy* array of *matplotlib* axes to plot on.
- **bar_kw** – Keyword arguments passed to the `matplotlib.pyplot.barh` function.
- **fig_kw** – Keyword arguments passed to the `matplotlib.figure.set` function.

Returns

plt.Axes with the feature importance plot.

Subpackages

alibi.explainers.anchors package

Submodules

alibi.explainers.anchors.anchor_base module

class `alibi.explainers.anchors.anchor_base.AnchorBaseBeam(samplers, **kwargs)`

Bases: `object`

__init__(`samplers, **kwargs`)

Parameters

samplers (`List[Callable]`) – Objects that can be called with args (`result, n_samples`) tuple to draw samples.

anchor_beam (`delta=0.05, epsilon=0.1, desired_confidence=1.0, beam_size=1, epsilon_stop=0.05, min_samples_start=100, max_anchor_size=None, stop_on_first=False, batch_size=100, coverage_samples=10000, verbose=False, verbose_every=1, **kwargs`)

Uses the KL-LUCB algorithm (Kaufmann and Kalyanakrishnan, 2013) together with additional sampling to search feature sets (anchors) that guarantee the prediction made by a classifier model. The search is greedy if `beam_size=1`. Otherwise, at each of the `max_anchor_size` steps, `beam_size` solutions are explored. By construction, solutions found have high precision (defined as the expected of number of times the classifier makes the same prediction when queried with the feature subset combined with arbitrary samples drawn from a noise distribution). The algorithm maximises the coverage of the solution found - the frequency of occurrence of records containing the feature subset in set of samples.

Parameters

- **delta** (`float`) – Used to compute *beta*.
- **epsilon** (`float`) – Precision bound tolerance for convergence.
- **desired_confidence** (`float`) – Desired level of precision (*tau* in [paper](#)).
- **beam_size** (`int`) – Beam width.
- **epsilon_stop** (`float`) – Confidence bound margin around desired precision.
- **min_samples_start** (`int`) – Min number of initial samples.
- **max_anchor_size** (`Optional[int]`) – Max number of features in result.
- **stop_on_first** (`bool`) – Stop on first valid result found.
- **coverage_samples** (`int`) – Number of samples from which to build a coverage set.
- **batch_size** (`int`) – Number of samples used for an arm evaluation.
- **verbose** (`bool`) – Whether to print intermediate LUCB & anchor selection output.
- **verbose_every** (`int`) – Print intermediate output every `verbose_every` steps.

Return type

`dict`

Returns

Explanation dictionary containing anchors with metadata like coverage and precision and examples.

static `compute_beta(n_features, t, delta)`

Parameters

- **n_features** (`int`) – Number of candidate anchors.
- **t** (`int`) – Iteration number.
- **delta** (`float`) – Confidence budget, candidate anchors have close to optimal precisions with prob. $1 - \delta$.

Return type

`float`

Returns

Level used to update upper and lower precision bounds.

static `dlow_bernoulli(p, level, n_iter=17)`

Update lower precision bound for a candidate anchors dependent on the KL-divergence.

Parameters

- **p** (`ndarray`) – Precision of candidate anchors.

- **level** (ndarray) – *beta / nb of samples* for each result.
- **n_iter** (int) – Number of iterations during lower bound update.

Return type
ndarray

Returns
Updated lower precision bounds array.

draw_samples(anchors, batch_size)

Parameters

- **anchors** (list) – Anchors on which samples are conditioned.
- **batch_size** (int) – The number of samples drawn for each result.

Return type
Tuple[tuple, tuple]

Returns
A tuple of positive samples (for which prediction matches desired label) and a tuple of total number of samples drawn.

static dup_bernoulli(p, level, n_iter=17)

Update upper precision bound for a candidate anchors dependent on the KL-divergence.

Parameters

- **p** (ndarray) – Precision of candidate anchors.
- **level** (ndarray) – *beta / nb of samples* for each result.
- **n_iter** (int) – Number of iterations during lower bound update.

Return type
ndarray

Returns
Updated upper precision bounds array.

get_anchor_metadata(features, success, batch_size=100)

Given the features contained in a result, it retrieves metadata such as the precision and coverage of the result and partial anchors and examples where the result/partial anchors apply and yield the same prediction as on the instance to be explained (*covered_true*) or a different prediction (*covered_false*).

Parameters

- **features** (tuple) – Sorted indices of features in result.
- **success** – Indicates whether an anchor satisfying precision threshold was met or not.
- **batch_size** (int) – Number of samples among which positive and negative examples for partial anchors are selected if partial anchors have not already been explicitly sampled.

Return type
dict

Returns
Anchor dictionary with result features and additional metadata.

get_init_stats(anchors, coverages=False)

Finds the number of samples already drawn for each result in anchors, their comparisons with the instance to be explained and, optionally, coverage.

Parameters

- **anchors** (`list`) – Candidate anchors.
- **coverages** – If True, the statistics returned contain the coverage of the specified anchors.

Return type`dict`**Returns**

Dictionary with lists containing nb of samples used and where sample predictions equal the desired label.

klucb(*anchors, init_stats, epsilon, delta, batch_size, top_n, verbose=False, verbose_every=1*)

Implements the KL-LUCB algorithm (Kaufmann and Kalyanakrishnan, 2013).

Parameters

- **anchors** (`list`) – A list of anchors from which two critical anchors are selected (see Kaufmann and Kalyanakrishnan, 2013).
- **init_stats** (`dict`) – Dictionary with lists containing nb of samples used and where sample predictions equal the desired label.
- **epsilon** (`float`) – Precision bound tolerance for convergence.
- **delta** (`float`) – Used to compute *beta*.
- **batch_size** (`int`) – Number of samples.
- **top_n** (`int`) – Min of beam width size or number of candidate anchors.
- **verbose** (`bool`) – Whether to print intermediate output.
- **verbose_every** (`int`) – Whether to print intermediate output every *verbose_every* steps.

Return type`ndarray`**Returns**

Indices of best result options. Number of indices equals min of beam width or nb of candidate anchors.

propose_anchors(*previous_best*)

Parameters

previous_best (`list`) – List with tuples of result candidates.

Return type`list`**Returns**

List with tuples of candidate anchors with additional metadata.

select_critical_arms(*means, ub, lb, n_samples, delta, top_n, t*)

Determines a set of two anchors by updating the upper bound for low empirical precision anchors and the lower bound for anchors with high empirical precision.

Parameters

- **means** (`ndarray`) – Empirical mean result precisions.
- **ub** (`ndarray`) – Upper bound on result precisions.
- **lb** (`ndarray`) – Lower bound on result precisions.

- **n_samples** (ndarray) – The number of samples drawn for each candidate result.
- **delta** (float) – Confidence budget, candidate anchors have close to optimal precisions with prob. $1 - \text{delta}$.
- **top_n** (int) – Number of arms to be selected.
- **t** (int) – Iteration number.

Returns

Upper and lower precision bound indices.

static to_sample(means, ubs, lbs, desired_confidence, epsilon_stop)

Given an array of mean result precisions and their upper and lower bounds, determines for which anchors more samples need to be drawn in order to estimate the anchors precision with *desired_confidence* and error tolerance.

Parameters

- **means** (ndarray) – Mean precisions (each element represents a different result).
- **ubs** (ndarray) – Precisions' upper bounds (each element represents a different result).
- **lbs** (ndarray) – Precisions' lower bounds (each element represents a different result).
- **desired_confidence** (float) – Desired level of confidence for precision estimation.
- **epsilon_stop** (float) – Tolerance around desired precision.

Returns

Boolean array indicating whether more samples are to be drawn for that particular result.

update_state(covered_true, covered_false, labels, samples, anchor)

Updates the explainer state (see [alibi.explainers.anchors.anchor_base.AnchorBaseBeam.__init__\(\)](#) for full state definition).

Parameters

- **covered_true** (ndarray) – Examples where the result applies and the prediction is the same as on the instance to be explained.
- **covered_false** (ndarray) – Examples where the result applies and the prediction is the different to the instance to be explained.
- **samples** (Tuple[ndarray, float]) – A tuple containing discretized data, coverage and the result sampled.
- **labels** (ndarray) – An array indicating whether the prediction on the sample matches the label of the instance to be explained.
- **anchor** (tuple) – The result to be updated.

Return type

`Tuple[int, int]`

Returns

A tuple containing the number of instances equals desired label of observation to be explained the total number of instances sampled, and the result that was sampled.

alibi.explainers.anchors.anchor_explanation module

class `alibi.explainers.anchors.anchor_explanation.AnchorExplanation`(*exp_type*, *exp_map*)

Bases: `object`

__init__(*exp_type*, *exp_map*)

Class used to unpack the anchors and metadata from the explainer dictionary.

Parameters

- **exp_type** (`str`) – Type of explainer: tabular, text or image.
- **exp_map** (`dict`) – Dictionary with the anchors and explainer metadata for an observation.

coverage(*partial_index=None*)

Parameters

partial_index (`Optional[int]`) – Get the result coverage until a certain index. For example, if the result has precisions `[0.1, 0.5, 0.95]` and `partial_index=1`, this will return `0.5`.

Return type

`float`

Returns

coverage – Anchor coverage.

examples(*only_different_prediction=False*, *only_same_prediction=False*, *partial_index=None*)

Parameters

- **only_different_prediction** (`bool`) – If True, will only return examples where the result makes a different prediction than the original model.
- **only_same_prediction** (`bool`) – If True, will only return examples where the result makes the same prediction than the original model.
- **partial_index** (`Optional[int]`) – Get the examples from the partial result until a certain index.

Return type

`Union[list, ndarray]`

Returns

Examples covered by result.

features(*partial_index=None*)

Parameters

partial_index (`Optional[int]`) – Get the result until a certain index. For example, if the result uses `segment_labels=(1, 2, 3)` and `partial_index=1`, this will return `[1, 2]`.

Return type

`list`

Returns

segment_labels – Features used in the result conditions.

names(*partial_index=None*)

Parameters

partial_index (`Optional[int]`) – Get the result until a certain index. For example, if the result is `(A=1, B=2, C=2)` and `partial_index=1`, this will return `["A=1", "B=2"]`.

Return type`list`**Returns***names* – Names with the result conditions.**precision**(*partial_index=None*)**Parameters****partial_index** (`Optional[int]`) – Get the result precision until a certain index. For example, if the result has precisions `[0.1, 0.5, 0.95]` and `partial_index=1`, this will return `0.5`.**Return type**`float`**Returns***precision* – Anchor precision.**alibi.explainers.anchors.anchor_image module**

```
class alibi.explainers.anchors.anchor_image.AnchorImage(predictor, image_shape, dtype=<class
    'numpy.float32'>, segmentation_fn='slic',
    segmentation_kwargs=None,
    images_background=None, seed=None)
```

Bases: `Explainer`

```
__init__(predictor, image_shape, dtype=<class 'numpy.float32'>, segmentation_fn='slic',
    segmentation_kwargs=None, images_background=None, seed=None)
```

Initialize anchor image explainer.

Parameters

- **predictor** (`Callable[[ndarray], ndarray]`) – A callable that takes a *numpy* array of *N* data points as inputs and returns *N* outputs.
- **image_shape** (`tuple`) – Shape of the image to be explained. The channel axis is expected to be last.
- **dtype** (`Type[generic]`) – A *numpy* scalar type that corresponds to the type of input array expected by *predictor*. This may be used to construct arrays of the given type to be passed through the *predictor*. For most use cases this argument should have no effect, but it is exposed for use with predictors that would break when called with an array of unsupported type.
- **segmentation_fn** (`Any`) – Any of the built in segmentation function strings: 'felzenszwalb', 'slic' or 'quickshift' or a custom segmentation function (callable) which returns an image mask with labels for each superpixel. The segmentation function is expected to return a segmentation mask containing all integer values from 0 to *K-1*, where *K* is the number of image segments (superpixels). See <http://scikit-image.org/docs/dev/api/skimage.segmentation.html> for more info.
- **segmentation_kwargs** (`Optional[dict]`) – Keyword arguments for the built in segmentation functions.
- **images_background** (`Optional[ndarray]`) – Images to overlay superpixels on.
- **seed** (`Optional[int]`) – If set, ensures different runs with the same input will yield same explanation.

Raises

- **`alibi.exceptions.PredictorCallError`** – If calling *predictor* fails at runtime.
- **`alibi.exceptions.PredictorReturnTypeError`** – If the return type of *predictor* is not `np.ndarray`.

```
explain(image, p_sample=0.5, threshold=0.95, delta=0.1, tau=0.15, batch_size=100,  
        coverage_samples=10000, beam_size=1, stop_on_first=False, max_anchor_size=None,  
        min_samples_start=100, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000,  
        verbose=False, verbose_every=1, **kwargs)
```

Explain instance and return anchor with metadata.

Parameters

- **`image`** (`ndarray`) – Image to be explained.
- **`p_sample`** (`float`) – The probability of simulating the absence of a superpixel. If the *images_background* is not provided, the absent superpixels will be replaced by the average value of their constituent pixels. Otherwise, the synthetic instances are created by fixing the present superpixels and superimposing another image from the *images_background* over the rest of the absent superpixels.
- **`threshold`** (`float`) – Minimum anchor precision threshold. The algorithm tries to find an anchor that maximizes the coverage under precision constraint. The precision constraint is formally defined as $P(\text{prec}(A) \geq t) \geq 1 - \delta$, where A is an anchor, t is the *threshold* parameter, δ is the *delta* parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, we are seeking for an anchor having its precision greater or equal than the given *threshold* with a confidence of $(1 - \text{delta})$. A higher value guarantees that the anchors are faithful to the model, but also leads to more computation time. Note that there are cases in which the precision constraint cannot be satisfied due to the quantile-based discretisation of the numerical features. If that is the case, the best (i.e. highest coverage) non-eligible anchor is returned.
- **`delta`** (`float`) – Significance threshold. $1 - \text{delta}$ represents the confidence threshold for the anchor precision (see *threshold*) and the selection of the best anchor candidate in each iteration (see *tau*).
- **`tau`** (`float`) – Multi-armed bandit parameter used to select candidate anchors in each iteration. The multi-armed bandit algorithm tries to find within a tolerance *tau* the most promising (i.e. according to the precision) *beam_size* candidate anchor(s) from a list of proposed anchors. Formally, when the *beam_size*=1, the multi-armed bandit algorithm seeks to find an anchor A such that $P(\text{prec}(A) \geq \text{prec}(A^*) - \tau) \geq 1 - \delta$, where A^* is the anchor with the highest true precision (which we don't know), τ is the *tau* parameter, δ is the *delta* parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, in each iteration, the algorithm returns with a probability of at least $1 - \text{delta}$ an anchor A with a precision within an error tolerance of *tau* from the precision of the highest true precision anchor A^* . A bigger value for *tau* means faster convergence but also looser anchor conditions.
- **`batch_size`** (`int`) – Batch size used for sampling. The Anchor algorithm will query the black-box model in batches of size *batch_size*. A larger *batch_size* gives more confidence in the anchor, again at the expense of computation time since it involves more model prediction calls.
- **`coverage_samples`** (`int`) – Number of samples used to estimate coverage from during result search.
- **`beam_size`** (`int`) – Number of candidate anchors selected by the multi-armed bandit algorithm in each iteration from a list of proposed anchors. A bigger beam width can lead to

a better overall anchor (i.e. prevents the algorithm of getting stuck in a local maximum) at the expense of more computation time.

- **stop_on_first** (`bool`) – If `True`, the beam search algorithm will return the first anchor that satisfies the probability constraint.
- **max_anchor_size** (`Optional[int]`) – Maximum number of features in result.
- **min_samples_start** (`int`) – Min number of initial samples.
- **n_covered_ex** (`int`) – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with *desired_label* are stored).
- **binary_cache_size** (`int`) – The result search pre-allocates *binary_cache_size* batches for storing the binary arrays returned during sampling.
- **cache_margin** (`int`) – When only `max(cache_margin, batch_size)` positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
- **verbose** (`bool`) – Display updates during the anchor search iterations.
- **verbose_every** (`int`) – Frequency of displayed iterations during anchor search process.

Return type

Explanation

Returns

explanation – *Explanation* object containing the anchor explaining the instance with additional metadata as attributes. See usage at [AnchorImage examples](#) for details.

generate_superpixels(*image*)

Generates superpixels from (i.e., segments) an image.

Parameters

image (`ndarray`) – A grayscale or RGB image.

Return type

`ndarray`

Returns

A $[H, W]$ array of integers. Each integer is a segment (superpixel) label.

overlay_mask(*image, segments, mask_features, scale=(0, 255)*)

Overlay image with mask described by the mask features.

Parameters

- **image** (`ndarray`) – Image to be explained.
- **segments** (`ndarray`) – Superpixels.
- **mask_features** (`list`) – List with superpixels present in mask.
- **scale** (`tuple`) – Pixel scale for masked image.

Return type

`ndarray`

Returns

masked_image – Image overlaid with mask.

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (*Callable*) – New predictor function.

Return type

None

class `alibi.explainers.anchors.anchor_image.AnchorImageSampler`(*predictor, segmentation_fn, custom_segmentation, image, images_background=None, p_sample=0.5, n_covered_ex=10*)

Bases: *object*

__call__(*anchor, num_samples, compute_labels=True*)

Sample images from a perturbation distribution by masking randomly chosen superpixels from the original image and replacing them with pixel values from superimposed images if background images are provided to the explainer. Otherwise, the superpixels from the original image are replaced with their average values.

Parameters

- **anchor** (*Tuple[int, tuple]*) –
 - *int* - order of anchor in the batch.
 - *tuple* - features (= superpixels) present in the proposed anchor.
- **num_samples** (*int*) – Number of samples used.
- **compute_labels** (*bool*) – If *True*, an array of comparisons between predictions on perturbed samples and instance to be explained is returned.

Return type

List[Union[ndarray, float, int]]

Returns

- If *compute_labels=True*, a list containing the following is returned –
 - *covered_true* - perturbed examples where the anchor applies and the model prediction on perturbed is the same as the instance prediction.
 - *covered_false* - perturbed examples where the anchor applies and the model prediction on perturbed sample is NOT the same as the instance prediction.
 - *labels* - *num_samples* ints indicating whether the prediction on the perturbed sample matches (1) the label of the instance to be explained or not (0).
 - *data* - Matrix with 1s and 0s indicating whether the values in a superpixel will remain unchanged (1) or will be perturbed (0), for each sample.
 - *-1.0* - indicates exact coverage is not computed for this algorithm.
 - *anchor[0]* - position of anchor in the batch request
- Otherwise, a list containing the data matrix only is returned.

__init__(*predictor, segmentation_fn, custom_segmentation, image, images_background=None, p_sample=0.5, n_covered_ex=10*)

Initialize anchor image sampler.

Parameters

- **predictor** (`Callable`) – A callable that takes a *numpy* array of N data points as inputs and returns N outputs.
- **segmentation_fn** (`Callable`) – Function used to segment the images. The segmentation function is expected to return a segmentation mask containing all integer values from 0 to $K-1$, where K is the number of image segments (superpixels).
- **image** (`ndarray`) – Image to be explained.
- **images_background** (`Optional[ndarray]`) – Images to overlay superpixels on.
- **p_sample** (`float`) – Probability for a pixel to be represented by the average value of its superpixel.
- **n_covered_ex** (`int`) – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with *desired_label* are stored).

compare_labels(*samples*)

Compute the agreement between a classifier prediction on an instance to be explained and the prediction on a set of samples which have a subset of perturbed superpixels.

Parameters

samples (`ndarray`) – Samples whose labels are to be compared with the instance label.

Return type

`ndarray`

Returns

A boolean array indicating whether the prediction was the same as the instance label.

generate_superpixels(*image*)

Generates superpixels from (i.e., segments) an image.

Parameters

image (`ndarray`) – A grayscale or RGB image.

Return type

`ndarray`

Returns

A $[H, W]$ array of integers. Each integer is a segment (superpixel) label.

perturbation(*anchor, num_samples*)

Perturbs an image by altering the values of selected superpixels. If a dataset of image backgrounds is provided to the explainer, then the superpixels are replaced with the equivalent superpixels from the background image. Otherwise, the superpixels are replaced by their average value.

Parameters

- **anchor** (`tuple`) – Contains the superpixels whose values are not going to be perturbed.
- **num_samples** (`int`) – Number of perturbed samples to be returned.

Return type

`Tuple[ndarray, ndarray]`

Returns

- *imgs* – A $[num_samples, H, W, C]$ array of perturbed images.
- *segments_mask* – A $[num_samples, M]$ binary mask, where M is the number of image superpixels segments. 1 indicates the values in that particular superpixels are not perturbed.

```
alibi.explainers.anchors.anchor_image.scale_image(image, scale=(0, 255))
```

Scales an image in a specified range.

Parameters

- **image** (ndarray) – Image to be scale.
- **scale** (tuple) – The scaling interval.

Return type

ndarray

Returns

img_scaled – Scaled image.

alibi.explainers.anchors.anchor_tabular module

```
class alibi.explainers.anchors.anchor_tabular.AnchorTabular(predictor, feature_names,
                                                             categorical_names=None,
                                                             dtype=<class 'numpy.float32'>,
                                                             ohe=False, seed=None)
```

Bases: [Explainer](#), [FitMixin](#)

```
__init__(predictor, feature_names, categorical_names=None, dtype=<class 'numpy.float32'>, ohe=False,
          seed=None)
```

Parameters

- **predictor** (Callable[[ndarray], ndarray]) – A callable that takes a *numpy* array of *N* data points as inputs and returns *N* outputs.
- **feature_names** (List[str]) – List with feature names.
- **categorical_names** (Optional[Dict[int, List[str]]]) – Dictionary where keys are feature columns and values are the categories for the feature.
- **dtype** (Type[generic]) – A *numpy* scalar type that corresponds to the type of input array expected by *predictor*. This may be used to construct arrays of the given type to be passed through the *predictor*. For most use cases this argument should have no effect, but it is exposed for use with predictors that would break when called with an array of unsupported type.
- **ohe** (bool) – Whether the categorical variables are one-hot encoded (OHE) or not. If not OHE, they are assumed to have ordinal encodings.
- **seed** (Optional[int]) – Used to set the random number generator for repeatability purposes.

Raises

- [alibi.exceptions.PredictorCallError](#) – If calling *predictor* fails at runtime.
- [alibi.exceptions.PredictorReturnTypeError](#) – If the return type of *predictor* is not *np.ndarray*.

```
add_names_to_exp(explanation)
```

Add feature names to explanation dictionary.

Parameters

explanation (dict) – Dict with anchors and additional metadata.

Return type

None

explain(*X*, *threshold*=0.95, *delta*=0.1, *tau*=0.15, *batch_size*=100, *coverage_samples*=10000, *beam_size*=1, *stop_on_first*=False, *max_anchor_size*=None, *min_samples_start*=100, *n_covered_ex*=10, *binary_cache_size*=10000, *cache_margin*=1000, *verbose*=False, *verbose_every*=1, ***kwargs*)

Explain prediction made by classifier on instance *X*.

Parameters

- **X** (ndarray) – Instance to be explained.
- **threshold** (float) – Minimum anchor precision threshold. The algorithm tries to find an anchor that maximizes the coverage under precision constraint. The precision constraint is formally defined as $P(\text{prec}(A) \geq t) \geq 1 - \delta$, where *A* is an anchor, *t* is the *threshold* parameter, δ is the *delta* parameter, and *prec*(·) denotes the precision of an anchor. In other words, we are seeking for an anchor having its precision greater or equal than the given *threshold* with a confidence of $(1 - \text{delta})$. A higher value guarantees that the anchors are faithful to the model, but also leads to more computation time. Note that there are cases in which the precision constraint cannot be satisfied due to the quantile-based discretisation of the numerical features. If that is the case, the best (i.e. highest coverage) non-eligible anchor is returned.
- **delta** (float) – Significance threshold. $1 - \text{delta}$ represents the confidence threshold for the anchor precision (see *threshold*) and the selection of the best anchor candidate in each iteration (see *tau*).
- **tau** (float) – Multi-armed bandit parameter used to select candidate anchors in each iteration. The multi-armed bandit algorithm tries to find within a tolerance *tau* the most promising (i.e. according to the precision) *beam_size* candidate anchor(s) from a list of proposed anchors. Formally, when the *beam_size*=1, the multi-armed bandit algorithm seeks to find an anchor *A* such that $P(\text{prec}(A) \geq \text{prec}(A^*) - \tau) \geq 1 - \delta$, where *A** is the anchor with the highest true precision (which we don't know), τ is the *tau* parameter, δ is the *delta* parameter, and *prec*(·) denotes the precision of an anchor. In other words, in each iteration, the algorithm returns with a probability of at least $1 - \text{delta}$ an anchor *A* with a precision within an error tolerance of *tau* from the precision of the highest true precision anchor *A**. A bigger value for *tau* means faster convergence but also looser anchor conditions.
- **batch_size** (int) – Batch size used for sampling. The Anchor algorithm will query the black-box model in batches of size *batch_size*. A larger *batch_size* gives more confidence in the anchor, again at the expense of computation time since it involves more model prediction calls.
- **coverage_samples** (int) – Number of samples used to estimate coverage from during result search.
- **beam_size** (int) – Number of candidate anchors selected by the multi-armed bandit algorithm in each iteration from a list of proposed anchors. A bigger beam width can lead to a better overall anchor (i.e. prevents the algorithm of getting stuck in a local maximum) at the expense of more computation time.
- **stop_on_first** (bool) – If True, the beam search algorithm will return the first anchor that has satisfies the probability constraint.
- **max_anchor_size** (Optional[int]) – Maximum number of features in result.
- **min_samples_start** (int) – Min number of initial samples.

- **n_covered_ex** (*int*) – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with *desired_label* are stored).
- **binary_cache_size** (*int*) – The result search pre-allocates *binary_cache_size* batches for storing the binary arrays returned during sampling.
- **cache_margin** (*int*) – When only $\max(\text{cache_margin}, \text{batch_size})$ positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
- **verbose** (*bool*) – Display updates during the anchor search iterations.
- **verbose_every** (*int*) – Frequency of displayed iterations during anchor search process.

Return type*Explanation***Returns**

explanation – *Explanation* object containing the result explaining the instance with additional metadata as attributes. See usage at [AnchorTabular examples](#) for details.

Raises

alibi.exceptions.NotFittedError – If *fit* has not been called prior to calling *explain*.

fit(*train_data*, *disc_perc*=(25, 50, 75), ***kwargs*)

Fit discretizer to train data to bin numerical features into ordered bins and compute statistics for numerical features. Create a mapping between the bin numbers of each discretised numerical feature and the row id in the training set where it occurs.

Parameters

- **train_data** (*ndarray*) – Representative sample from the training data.
- **disc_perc** (*Tuple[Union[int, float], ...]*) – List with percentiles (*int*) used for discretization.

Return type*AnchorTabular*

instance_label: *int*

The label of the instance to be explained.

meta: *dict*

Object metadata.

property predictor: *Callable* | *None*

Return type*Optional[Callable]*

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (*Callable*) – New predictor function.

Return type*None*

samplers: *list*


```
class alibi.explainers.anchors.anchor_tabular.TabularSampler(predictor, disc_perc,
                                                            numerical_features,
                                                            categorical_features, feature_names,
                                                            feature_values, n_covered_ex=10,
                                                            seed=None)
```

Bases: `object`

A sampler that uses an underlying training set to draw records that have a subset of features with values specified in an instance to be explained, X .

`__call__`(*anchor*, *num_samples*, *compute_labels=True*)

Obtain perturbed records by drawing samples from training data that contain the categorical labels and discretized numerical features and replacing the remainder of the record with arbitrary values.

Parameters

- **anchor** (`Tuple[int, tuple]`) – The integer represents the order of the result in a request array. The tuple contains encoded feature indices.
- **num_samples** (`int`) – Number of samples used when sampling from training set.
- **compute_labels** – If `True`, an array of comparisons between predictions on perturbed samples and instance to be explained is returned.

Return type

`Union[List[Union[ndarray, float, int]], List[ndarray]]`

Returns

- If `compute_labels=True`, a list containing the following is returned –
 - *covered_true* - perturbed examples where the anchor applies and the model prediction on perturbation is the same as the instance prediction.
 - *covered_false* - perturbed examples where the anchor applies and the model prediction is NOT the same as the instance prediction.
 - *labels* - *num_samples* ints indicating whether the prediction on the perturbed sample matches (1) the label of the instance to be explained or not (0).
 - *data* - Sampled data where ordinal features are binned (1 if in bin, 0 otherwise).
 - *coverage* - the coverage of the anchor.
 - *anchor[0]* - position of anchor in the batch request.
- Otherwise, a list containing the data matrix only is returned.

`__init__`(*predictor*, *disc_perc*, *numerical_features*, *categorical_features*, *feature_names*, *feature_values*, *n_covered_ex=10*, *seed=None*)

Parameters

- **predictor** (`Callable`) – A callable that takes a tensor of N data points as inputs and returns N outputs.
- **disc_perc** (`Tuple[Union[int, float], ...]`) – Percentiles used for numerical feature discretisation.
- **numerical_features** (`List[int]`) – Numerical features column IDs.
- **categorical_features** (`List[int]`) – Categorical features column IDs.
- **feature_names** (`list`) – Feature names.

- **feature_values** (`dict`) – Key: categorical feature column ID, value: values for the feature.
- **n_covered_ex** (`int`) – For each result, a number of samples where the prediction agrees/disagrees with the prediction on instance to be explained are stored.
- **seed** (`Optional[int]`) – If set, fixes the random number sequence.

build_lookups(*X*)

An encoding of the feature IDs is created by assigning each bin of a discretized numerical variable and each categorical variable a unique index. For a dataset containing, e.g., a numerical variable with 5 bins and 3 categorical variables, indices 0 - 4 represent bins of the numerical variable whereas indices 5, 6, 7 represent the encoded indices of the categorical variables (but see note for caviats). The encoding is necessary so that the different ranges of the numerical variable can be sampled during result construction. Note that the encoded indices represent the predicates used during the anchor construction process (i.e., and anchor is a collection of encoded indices).

Parameters

X (`ndarray`) – Instance to be explained.

Return type

`List[Dict]`

Returns

A list containing three dictionaries, whose keys are encoded feature IDs –

- *cat_lookup* - maps categorical variables to their value in *X*.
- *ord_lookup* - maps discretized numerical variables to the bins they can be sampled from given *X*.
- *enc2feat_idx* - maps the encoded IDs to the original (training set) feature column IDs.

Notes

Each continuous variable has *n_bins - 1* corresponding entries in *ord_lookup*.

compare_labels(*samples*)

Compute the agreement between a classifier prediction on an instance to be explained and the prediction on a set of samples which have a subset of features fixed to specific values.

Parameters

samples (`ndarray`) – Samples whose labels are to be compared with the instance label.

Return type

`ndarray`

Returns

An array of integers indicating whether the prediction was the same as the instance label.

deferred_init(*train_data*, *d_train_data*)

Initialise the tabular sampler object with data, discretizer, feature statistics and build an index from feature values and bins to database rows for each feature.

Parameters

- **train_data** (`Union[ndarray, Any]`) – Data from which samples are drawn. Can be a *numpy* array or a *ray* future.
- **d_train_data** (`Union[ndarray, Any]`) – Discretized version for training data. Can be a *numpy* array or a *ray* future.

Return type*Any***Returns***An initialised sampler.***get_features_index(anchor)**

Given an anchor, this function finds the row indices in the training set where the feature has the same value as the feature in the instance to be explained (for ordinal variables, the row indices are those of rows which contain records with feature values in the same bin). The algorithm uses both the feature *encoded* ids in anchor and the feature ids in the input data set. The two are mapped by *self.enc2feat_idx*.

Parameters

anchor (*tuple*) – The anchor for which the training set row indices are to be retrieved. The ints represent encoded feature ids.

Return type*Tuple[Dict[int, Set[int]], Dict[int, Any], List[Tuple[int, str, Union[Any, int]]]]***Returns**

- *allowed_bins* – Maps original feature ids to the bins that the feature should be sampled from given the input anchor.
- *allowed_rows* – Maps original feature ids to the training set rows where these features have the same value as the anchor.
- *unk_feat_values* – When a categorical variable with the specified value/discretized variable in the specified bin is not found in the training set, a tuple is added to *unk_feat_values* to indicate the original feature id, its type ('c' = categorical, 'o' = discretized continuous) and the value/bin it should be sampled from.

handle_unk_features(allowed_bins, num_samples, samples, unk_feature_values)

Replaces unknown feature values with defaults. For categorical variables, the replacement value is the same as the value of the unknown feature. For continuous variables, a value is sampled uniformly at random from the feature range.

Parameters

- **allowed_bins** (*Dict[int, Set[int]]*) – See [alibi.explainers.anchors.anchor_tabular.TabularSampler.get_features_index\(\)](#) method.
- **num_samples** (*int*) – Number of replacement values.
- **samples** (*ndarray*) – Contains the samples whose values are to be replaced.
- **unk_feature_values** (*List[Tuple[int, str, Union[Any, int]]]*) – List of tuples where: [0] is original feature id, [1] feature type, [2] if var is categorical, replacement value, otherwise None

Return type*None***instance_label: int**

The label of the instance to be explained.

perturbation(anchor, num_samples)

Implements functionality described in [alibi.explainers.anchors.anchor_tabular.TabularSampler.__call__\(\)](#).

Parameters

- **anchor** (*tuple*) – Each int is an encoded feature id.

- **num_samples** (*int*) – Number of samples.

Return type

`Tuple[ndarray, ndarray, float]`

Returns

- *samples* – Sampled data from training set.
- *d_samples* – Like *samples*, but continuous data is converted to ordinal discrete data (binned).
- *coverage* – The coverage of the result in the training data.

replace_features(*samples, allowed_rows, uniq_feat_ids, partial_anchor_rows, nb_partial_anchors, num_samples*)

The method creates perturbed samples by first replacing all partial anchors with partial anchors drawn from the training set. Then remainder of the features are then replaced with random values drawn from the same bin for discretized continuous features and same value for categorical features.

Parameters

- **samples** (*ndarray*) – Randomly drawn samples, where the anchor does not apply.
- **allowed_rows** (*Dict[int, Any]*) – Maps feature ids to the rows indices in training set where the feature has same value as instance (cat.) or is in the same bin.
- **uniq_feat_ids** (*List[int]*) – Multiple encoded features in the anchor can map to the same original feature id. Unique features in the anchor. This is the list of unique original features id in the anchor.
- **partial_anchor_rows** (*List[ndarray]*) – The rows in the training set where each partial anchor applies. Last entry is an array of row indices where the entire anchor applies.
- **nb_partial_anchors** (*ndarray*) – The number of training records which contain each partial anchor.
- **num_samples** (*int*) – Number of perturbed samples to be returned.

Return type

`None`

set_instance_label(*X*)

Sets the sampler label. Necessary for setting the remote sampling process state during explain call.

Parameters

X (*ndarray*) – Instance to be explained.

Return type

`None`

set_n_covered(*n_covered*)

Set the number of examples to be saved for each result and partial result during search process. The same number of examples is saved in the case where the predictions on perturbed samples and original instance agree or disagree.

Parameters

n_covered (*int*) – Number of examples to be saved.

Return type

`None`

alibi.explainers.anchors.anchor_tabular_distributed module

class `alibi.explainers.anchors.anchor_tabular_distributed.DistributedAnchorBaseBeam`(*samplers*,
***kwargs*)

Bases: `AnchorBaseBeam`

draw_samples(*anchors*, *batch_size*)

Distributes sampling requests among processes running sampling tasks.

Parameters

- **anchors** (list) – See `alibi.explainers.anchors.anchor_base.AnchorBaseBeam.draw_samples()` implementation.
- **batch_size** (int) – See `alibi.explainers.anchors.anchor_base.AnchorBaseBeam.draw_samples()` implementation.

Return type

`Tuple[ndarray, ndarray]`

Returns

See `alibi.explainers.anchors.anchor_base.AnchorBaseBeam.draw_samples()` implementation.

class `alibi.explainers.anchors.anchor_tabular_distributed.DistributedAnchorTabular`(*predictor*,
feature_names,
category_names,
cal_names=None,
dtype=<class
'numpy.float32'>,
one=False,
seed=None)

Bases: `AnchorTabular`

explain(*X*, *threshold*=0.95, *delta*=0.1, *tau*=0.15, *batch_size*=100, *coverage_samples*=10000, *beam_size*=1,
stop_on_first=False, *max_anchor_size*=None, *min_samples_start*=1, *n_covered_ex*=10,
binary_cache_size=10000, *cache_margin*=1000, *verbose*=False, *verbose_every*=1, ***kwargs*)

Explains the prediction made by a classifier on instance *X*. Sampling is done in parallel over a number of cores specified in *kwargs*['*ncpu*'].

Parameters

- **X** (ndarray) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **threshold** (float) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **delta** (float) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **tau** (float) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **batch_size** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.

- **coverage_samples** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **beam_size** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **stop_on_first** (bool) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **max_anchor_size** (Optional[int]) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **min_samples_start** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **n_covered_ex** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **binary_cache_size** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **cache_margin** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **verbose** (bool) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- **verbose_every** (int) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.
- ****kwargs** (Any) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()`.

Return type

Explanation

Returns

See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.explain()` superclass.

fit(*train_data*, *disc_perc*=(25, 50, 75), ***kwargs*)

Creates a list of handles to parallel processes handles that are used for submitting sampling tasks.

Parameters

- **train_data** (ndarray) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.fit()` superclass.
- **disc_perc** (tuple) – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.fit()` superclass.
- ****kwargs** – See `alibi.explainers.anchors.anchor_tabular.AnchorTabular.fit()` superclass.

Return type

AnchorTabular

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (Callable) – New model prediction function.

Return type

None

class `alibi.explainers.anchors.anchor_tabular_distributed.RemoteSampler(*args)`

Bases: `object`

A wrapper that facilitates the use of *TabularSampler* for distributed sampling.

__call__(*anchors_batch*, *num_samples*, *compute_labels=True*)

Wrapper around `alibi.explainers.anchors.anchor_tabular.TabularSampler.__call__()`. It allows sampling a batch of anchors in the same process, which can improve performance.

Parameters

- **anchors_batch** (`Union[Tuple[int, tuple], List[Tuple[int, tuple]]]`) – A list of result tuples. See `alibi.explainers.anchors.anchor_tabular.TabularSampler.__call__()` for details.
- **num_samples** (`int`) – A list of result tuples. See `alibi.explainers.anchors.anchor_tabular.TabularSampler.__call__()` for details.
- **compute_labels** (`bool`) – A list of result tuples. See `alibi.explainers.anchors.anchor_tabular.TabularSampler.__call__()` for details.

Return type

List

build_lookups(*X*)

Wrapper around `alibi.explainers.anchors.anchor_tabular.TabularSampler.build_lookups()`.

Parameters

X (`ndarray`) – See `alibi.explainers.anchors.anchor_tabular.TabularSampler.build_lookups()`.

Returns

See `alibi.explainers.anchors.anchor_tabular.TabularSampler.build_lookups()`.

set_instance_label(*X*)

Sets the remote sampler instance label.

Parameters

X (`ndarray`) – The instance to be explained.

Return type

int

Returns

label – The label of the instance to be explained.

set_n_covered(*n_covered*)

Sets the remote sampler number of examples to save for inspection.

Parameters

n_covered (`int`) – Number of examples where the result (and partial anchors) apply.

Return type

None

alibi.explainers.anchors.anchor_text module

```
class alibi.explainers.anchors.anchor_text.AnchorText(predictor, sampling_strategy='unknown',
                                                       nlp=None, language_model=None, seed=0,
                                                       **kwargs)
```

Bases: [Explainer](#)

```
CLASS_SAMPLER = {'language_model': <class
'alibi.explainers.anchors.language_model_text_sampler.LanguageModelSampler'>,
'similarity': <class 'alibi.explainers.anchors.text_samplers.SimilaritySampler'>,
'unknown': <class 'alibi.explainers.anchors.text_samplers.UnknownSampler'>}
```

```
DEFAULTS: Dict[str, Dict] = {'language_model': {'batch_size_lm': 32, 'filling':
'parallel', 'frac_mask_templates': 0.1, 'punctuation':
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~', 'sample_proba': 0.5, 'sample_punctuation':
False, 'stopwords': [], 'temperature': 1.0, 'top_n': 100, 'use_proba': False},
'similarity': {'sample_proba': 0.5, 'temperature': 1.0, 'top_n': 100, 'use_proba':
False}, 'unknown': {'sample_proba': 0.5}}
```

```
SAMPLING_LANGUAGE_MODEL = 'language_model'
```

Language model sampling strategy.

```
SAMPLING_SIMILARITY = 'similarity'
```

Similarity sampling strategy.

```
SAMPLING_UNKNOWN = 'unknown'
```

Unknown sampling strategy.

```
__init__(predictor, sampling_strategy='unknown', nlp=None, language_model=None, seed=0, **kwargs)
```

Initialize anchor text explainer.

Parameters

- **predictor** ([Callable](#)[\[\[List\[str\]\], ndarray\]](#)) – A callable that takes a list of text strings representing N data points as inputs and returns N outputs.
- **sampling_strategy** ([str](#)) – Perturbation distribution method:
 - 'unknown' - replaces words with UNKs.
 - 'similarity' - samples according to a similarity score with the corpus embeddings.
 - 'language_model' - samples according the language model's output distributions.
- **nlp** ([Optional](#)[\[Language\]](#)) – *spaCy* object when sampling method is 'unknown' or 'similarity'.
- **language_model** ([Optional](#)[\[LanguageModel\]](#)) – Transformers masked language model. This is a model that it adheres to the *LanguageModel* interface we define in [alibi.utils.lang_model.LanguageModel](#).
- **seed** ([int](#)) – If set, ensure identical random streams.
- **kwargs** ([Any](#)) – Sampling arguments can be passed as *kwargs* depending on the *sampling_strategy*. Check default arguments defined in:
 - `alibi.explainers.anchor_text.DEFAULT_SAMPLING_UNKNOWN`
 - `alibi.explainers.anchor_text.DEFAULT_SAMPLING_SIMILARITY`
 - `alibi.explainers.anchor_text.DEFAULT_SAMPLING_LANGUAGE_MODEL`

Raises

- **`alibi.exceptions.PredictorCallError`** – If calling *predictor* fails at runtime.
- **`alibi.exceptions.PredictorReturnTypeError`** – If the return type of *predictor* is not `np.ndarray`.

`compare_labels(samples)`

Compute the agreement between a classifier prediction on an instance to be explained and the prediction on a set of samples which have a subset of features fixed to a given value (aka compute the precision of anchors).

Parameters

`samples` (`ndarray`) – Samples whose labels are to be compared with the instance label.

Return type

`ndarray`

Returns

A *numpy* boolean array indicating whether the prediction was the same as the instance label.

`explain(text, threshold=0.95, delta=0.1, tau=0.15, batch_size=100, coverage_samples=10000, beam_size=1, stop_on_first=True, max_anchor_size=None, min_samples_start=100, n_covered_ex=10, binary_cache_size=10000, cache_margin=1000, verbose=False, verbose_every=1, **kwargs)`

Explain instance and return anchor with metadata.

Parameters

- **`text`** (`str`) – Text instance to be explained.
- **`threshold`** (`float`) – Minimum anchor precision threshold. The algorithm tries to find an anchor that maximizes the coverage under precision constraint. The precision constraint is formally defined as $P(\text{prec}(A) \geq t) \geq 1 - \delta$, where A is an anchor, t is the *threshold* parameter, δ is the *delta* parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, we are seeking for an anchor having its precision greater or equal than the given *threshold* with a confidence of $(1 - \text{delta})$. A higher value guarantees that the anchors are faithful to the model, but also leads to more computation time. Note that there are cases in which the precision constraint cannot be satisfied due to the quantile-based discretisation of the numerical features. If that is the case, the best (i.e. highest coverage) non-eligible anchor is returned.
- **`delta`** (`float`) – Significance threshold. $1 - \text{delta}$ represents the confidence threshold for the anchor precision (see *threshold*) and the selection of the best anchor candidate in each iteration (see *tau*).
- **`tau`** (`float`) – Multi-armed bandit parameter used to select candidate anchors in each iteration. The multi-armed bandit algorithm tries to find within a tolerance *tau* the most promising (i.e. according to the precision) *beam_size* candidate anchor(s) from a list of proposed anchors. Formally, when the *beam_size*=1, the multi-armed bandit algorithm seeks to find an anchor A such that $P(\text{prec}(A) \geq \text{prec}(A^*) - \tau) \geq 1 - \delta$, where A^* is the anchor with the highest true precision (which we don't know), τ is the *tau* parameter, δ is the *delta* parameter, and $\text{prec}(\cdot)$ denotes the precision of an anchor. In other words, in each iteration, the algorithm returns with a probability of at least $1 - \text{delta}$ an anchor A with a precision within an error tolerance of *tau* from the precision of the highest true precision anchor A^* . A bigger value for *tau* means faster convergence but also looser anchor conditions.
- **`batch_size`** (`int`) – Batch size used for sampling. The Anchor algorithm will query the black-box model in batches of size *batch_size*. A larger *batch_size* gives more confidence

in the anchor, again at the expense of computation time since it involves more model prediction calls.

- **coverage_samples** (`int`) – Number of samples used to estimate coverage from during anchor search.
- **beam_size** (`int`) – Number of candidate anchors selected by the multi-armed bandit algorithm in each iteration from a list of proposed anchors. A bigger beam width can lead to a better overall anchor (i.e. prevents the algorithm of getting stuck in a local maximum) at the expense of more computation time.
- **stop_on_first** (`bool`) – If `True`, the beam search algorithm will return the first anchor that has satisfies the probability constraint.
- **max_anchor_size** (`Optional[int]`) – Maximum number of features to include in an anchor.
- **min_samples_start** (`int`) – Number of samples used for anchor search initialisation.
- **n_covered_ex** (`int`) – How many examples where anchors apply to store for each anchor sampled during search (both examples where prediction on samples agrees/disagrees with predicted label are stored).
- **binary_cache_size** (`int`) – The anchor search pre-allocates *binary_cache_size* batches for storing the boolean arrays returned during sampling.
- **cache_margin** (`int`) – When only `max(cache_margin, batch_size)` positions in the binary cache remain empty, a new cache of the same size is pre-allocated to continue buffering samples.
- **verbose** (`bool`) – Display updates during the anchor search iterations.
- **verbose_every** (`int`) – Frequency of displayed iterations during anchor search process.
- ****kwargs** (`Any`) – Other keyword arguments passed to the anchor beam search and the text sampling and perturbation functions.

Return type

Explanation

Returns

Explanation object containing the anchor explaining the instance with additional metadata as attributes. Contains the following data-related attributes –

- *anchor* : `List[str]` - a list of words in the proposed anchor.
- *precision* : `float` - the fraction of times the sampled instances where the anchor holds yields the same prediction as the original instance. The precision will always be threshold for a valid anchor.
- *coverage* : `float` - the fraction of sampled instances the anchor applies to.

meta: `dict`

Object metadata.

model: `spacy.language.Language` | *LanguageModel*

Language model to be used.

perturbation: `Any`

Perturbation method.

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (*Callable*) – New predictor function.

Return type

None

sampler(*anchor*, *num_samples*, *compute_labels=True*)

Generate perturbed samples while maintaining features in positions specified in anchor unchanged.

Parameters

- **anchor** (*Tuple[int, tuple]*) –
 - *int* - the position of the anchor in the input batch.
 - *tuple* - the anchor itself, a list of words to be kept unchanged.
- **num_samples** (*int*) – Number of generated perturbed samples.
- **compute_labels** (*bool*) – If True, an array of comparisons between predictions on perturbed samples and instance to be explained is returned.

Return type

Union[List[Union[ndarray, float, int]], List[ndarray]]

Returns

- If *compute_labels=True*, a list containing the following is returned –
 - *covered_true* - perturbed examples where the anchor applies and the model prediction on perturbation is the same as the instance prediction.
 - *covered_false* - perturbed examples where the anchor applies and the model prediction is NOT the same as the instance prediction.
 - *labels* - *num_samples* ints indicating whether the prediction on the perturbed sample matches (1) the label of the instance to be explained or not (0).
 - *data* - Matrix with 1s and 0s indicating whether a word in the text has been perturbed for each sample.
 - *-1.0* - indicates exact coverage is not computed for this algorithm.
 - *anchor[0]* - position of anchor in the batch request.
- Otherwise, a list containing the data matrix only is returned.

```
alibi.explainers.anchors.anchor_text.DEFAULT_SAMPLING_LANGUAGE_MODEL = {'batch_size_lm':
32, 'filling': 'parallel', 'frac_mask_templates': 0.1, 'punctuation':
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~', 'sample_proba': 0.5, 'sample_punctuation': False,
'stopwords': [], 'temperature': 1.0, 'top_n': 100, 'use_proba': False}
```

Default perturbation options for 'language_model' sampling

- 'filling' : str - filling method for language models. Allowed values: 'parallel', 'autoregressive'. 'parallel' method corresponds to a single forward pass through the language model. The masked words are sampled independently, according to the selected probability distribution (see *top_n*, *temperature*, *use_proba*). *autoregressive* method fills the words one at the time. This corresponds to multiple forward passes through the language model which is computationally expensive.
- 'sample_proba' : float - probability of a word to be masked.
- 'top_n' : int - number of similar words to sample for perturbations.

- 'temperature' : float - sample weight hyper-parameter if use_proba equals True.
- 'use_proba' : bool - whether to sample according to the predicted words distribution. If set to False, the *top_n* words are sampled uniformly at random.
- 'frac_mask_template' : float - fraction from the number of samples of mask templates to be generated. In each sampling call, will generate $\text{int}(\text{frac_mask_templates} * \text{num_samples})$ masking templates. Lower fraction corresponds to lower computation time since the batch fed to the language model is smaller. After the words' distributions is predicted for each mask, a total of *num_samples* will be generated by sampling evenly from each template. Note that lower fraction might correspond to less diverse sample. A *sample_proba=1* corresponds to masking each word. For this case only one masking template will be constructed. A *filling='autoregressive'* will generate *num_samples* masking templates regardless of the value of *frac_mask_templates*.
- 'batch_size_lm' : int - batch size used for the language model forward pass.
- 'punctuation' : str - string of punctuation not to be masked.
- 'stopwords' : List[str] - list of words not to be masked.
- 'sample_punctuation' : bool - whether to sample punctuation to fill the masked words. If False, the punctuation defined in *punctuation* will not be sampled.

```
alibi.explainers.anchors.anchor_text.DEFAULT_SAMPLING_SIMILARITY = {'sample_proba': 0.5,  
'temperature': 1.0, 'top_n': 100, 'use_proba': False}
```

Default perturbation options for 'similarity' sampling

- 'sample_proba' : float - probability of a word to be masked.
- 'top_n' : int - number of similar words to sample for perturbations.
- 'temperature' : float - sample weight hyper-parameter if *use_proba=True*.
- 'use_proba' : bool - whether to sample according to the words similarity.

```
alibi.explainers.anchors.anchor_text.DEFAULT_SAMPLING_UNKNOWN = {'sample_proba': 0.5}
```

Default perturbation options for 'unknown' sampling

- 'sample_proba' : float - probability of a word to be masked.

alibi.explainers.anchors.language_model_text_sampler module

```
class alibi.explainers.anchors.language_model_text_sampler.LanguageModelSampler(model, per-  
turb_opts)
```

Bases: [AnchorTextSampler](#)

FILLING_AUTOREGRESSIVE = 'autoregressive'

Autoregressive filling procedure. Considerably slow.

FILLING_PARALLEL: str = 'parallel'

Parallel filling procedure.

__call__(anchor, num_samples)

The function returns a *numpy* array of *num_samples* where randomly chosen features, except those in anchor, are replaced by words sampled according to the language model's predictions.

Parameters

- **anchor** (tuple) – Indices represent the positions of the words to be kept unchanged.
- **num_samples** (int) – Number of perturbed sentences to be returned.

Return type`Tuple[ndarray, ndarray]`**Returns**

See `alibi.explainers.anchors.language_model_text_sampler.LanguageModelSampler.perturb_sentence()`.

`__init__(model, perturb_opts)`

Initialize language model sampler. This sampler replaces words with the ones sampled according to the output distribution of the language model. There are two modes to use the sampler: 'parallel' and 'autoregressive'. In the 'parallel' mode, all words are replaced simultaneously. In the 'autoregressive' model, the words are replaced one by one, starting from left to right. Thus the following words are conditioned on the previous predicted words.

Parameters

- **model** (`LanguageModel`) – Transformers masked language model.
- **perturb_opts** (`dict`) – Perturbation options.

`create_mask(anchor, num_samples, sample_proba=1.0, filling='parallel', frac_mask_templates=0.1, **kwargs)`

Create mask for words to be perturbed.

Parameters

- **anchor** (`tuple`) – Indices represent the positions of the words to be kept unchanged.
- **num_samples** (`int`) – Number of perturbed sentences to be returned.
- **sample_proba** (`float`) – Probability of a word being replaced.
- **filling** (`str`) – Method to fill masked words. Either 'parallel' or 'autoregressive'.
- **frac_mask_templates** (`float`) – Fraction of mask templates from the number of requested samples.
- ****kwargs** – Other arguments to be passed to other methods.

Return type`Tuple[ndarray, ndarray]`**Returns**

- *raw* – Array with masked instances.
- *data* – A (*num_samples*, *m*)-dimensional boolean array, where *m* is the number of tokens in the instance to be explained.

`fill_mask(raw, data, num_samples, top_n=100, batch_size_lm=32, filling='parallel', **kwargs)`

Fill in the masked tokens with language model.

Parameters

- **raw** (`ndarray`) – Array of mask templates.
- **data** (`ndarray`) – Binary mask having 0 where the word was masked.
- **num_samples** (`int`) – Number of samples to be drawn.
- **top_n** (`int`) – Use the top n words when sampling.
- **batch_size_lm** (`int`) – Batch size used for language model.

- **filling** (`str`) – Method to fill masked words. Either 'parallel' or 'autoregressive'.
- ****kwargs** – Other parameters to be passed to other methods.

Return type`Tuple[ndarray, ndarray]`**Returns**

raw – Array containing *num_samples* elements. Each element is a perturbed sentence.

get_sample_ids(*punctuation='!\"#\$%&\\()*+,-./:;<=>?@[\\]^_`{|}~'*, *stopwords=None*, ***kwargs*)

Find indices in words which can be perturbed.

Parameters

- **punctuation** (`str`) – String of punctuation characters.
- **stopwords** (`Optional[List[str]]`) – List of stopwords.
- ****kwargs** – Other arguments. Not used.

Return type`None`

perturb_sentence(*anchor*, *num_samples*, *sample_proba=0.5*, *top_n=100*, *batch_size_lm=32*, *filling='parallel'*, ***kwargs*)

The function returns an *numpy* array of *num_samples* where randomly chosen features, except those in *anchor*, are replaced by words sampled according to the language model's predictions.

Parameters

- **anchor** (`tuple`) – Indices represent the positions of the words to be kept unchanged.
- **num_samples** (`int`) – Number of perturbed sentences to be returned.
- **sample_proba** (`float`) – Probability of a token being replaced by a similar token.
- **top_n** (`int`) – Used for top n sampling.
- **batch_size_lm** (`int`) – Batch size used for language model.
- **filling** (`str`) – Method to fill masked words. Either 'parallel' or 'autoregressive'.
- ****kwargs** – Other arguments to be passed to other methods.

Return type`Tuple[ndarray, ndarray]`**Returns**

- *raw* – Array containing *num_samples* elements. Each element is a perturbed sentence.
- *data* – A (*num_samples*, *m*)-dimensional boolean array, where *m* is the number of tokens in the instance to be explained.

seed(*seed*)

Return type`None`

set_data_type()

Working with *numpy* arrays of strings requires setting the data type to avoid truncating examples. This function estimates the longest sentence expected during the sampling process, which is used to set the number of characters for the samples and examples arrays. This depends on the perturbation method used for sampling.

Return type

None

set_text(text)

Sets the text to be processed

Parameters

text (*str*) – Text to be processed.

Return type

None

alibi.explainers.anchors.text_samplers module**class alibi.explainers.anchors.text_samplers.AnchorTextSampler**

Bases: *object*

abstract set_text(text)**Return type**

None

class alibi.explainers.anchors.text_samplers.Neighbors(nlp_obj, n_similar=500, w_prob=-15.0)

Bases: *object*

__init__(nlp_obj, n_similar=500, w_prob=-15.0)

Initialize class identifying neighbouring words from the embedding for a given word.

Parameters

- **nlp_obj** (*Language*) – *spaCy* model.
- **n_similar** (*int*) – Number of similar words to return.
- **w_prob** (*float*) – Smoothed log probability estimate of token's type.

neighbors(word, tag, top_n)

Find similar words for a certain word in the vocabulary.

Parameters

- **word** (*str*) – Word for which we need to find similar words.
- **tag** (*str*) – Part of speech tag for the words.
- **top_n** (*int*) – Return only *top_n* neighbors.

Return type

dict

Returns

A dict with two fields. The 'words' field contains a *numpy* array of the *top_n* most similar words, whereas the fields 'similarities' is a *numpy* array with corresponding word similarities.

```
class alibi.explainers.anchors.text_samplers.SimilaritySampler(nlp, perturb_opts)
```

Bases: [AnchorTextSampler](#)

```
__call__(anchor, num_samples)
```

The function returns a *numpy* array of *num_samples* where randomly chosen features, except those in anchor, are replaced by similar words with the same part of speech of tag. See [alibi.explainers.anchors.text_samplers.SimilaritySampler.perturb_sentence_similarity\(\)](#) for details of how the replacement works.

Parameters

- **anchor** ([tuple](#)) – Indices represent the positions of the words to be kept unchanged.
- **num_samples** ([int](#)) – Number of perturbed sentences to be returned.

Return type

[Tuple](#)[[ndarray](#), [ndarray](#)]

Returns

See [alibi.explainers.anchors.text_samplers.SimilaritySampler.perturb_sentence_similarity\(\)](#).

```
__init__(nlp, perturb_opts)
```

Initialize similarity sampler. This sampler replaces words with similar words.

Parameters

- **nlp** ([Language](#)) – *spaCy* object.
- **perturb_opts** ([Dict](#)) – Perturbation options.

```
find_similar_words()
```

This function queries a *spaCy* nlp model to find *n* similar words with the same part of speech for each word in the instance to be explained. For each word the search procedure returns a dictionary containing a *numpy* array of words ('words') and a *numpy* array of word similarities ('similarities').

Return type

[None](#)

```
perturb_sentence_similarity(present, n, sample_proba=0.5, forbidden=frozenset({}),
                           forbidden_tags=frozenset({'PRP$'}), forbidden_words=frozenset({'be'}),
                           temperature=1.0, pos=frozenset({'ADJ', 'ADP', 'ADV', 'DET', 'NOUN',
                                                           'VERB'}), use_proba=False, **kwargs)
```

Perturb the text instance to be explained.

Parameters

- **present** ([tuple](#)) – Word index in the text for the words in the proposed anchor.
- **n** ([int](#)) – Number of samples used when sampling from the corpus.
- **sample_proba** ([float](#)) – Sample probability for a word if *use_proba=False*.
- **forbidden** ([frozenset](#)) – Forbidden lemmas.
- **forbidden_tags** ([frozenset](#)) – Forbidden POS tags.
- **forbidden_words** ([frozenset](#)) – Forbidden words.
- **pos** ([frozenset](#)) – POS that can be changed during perturbation.
- **use_proba** ([bool](#)) – Bool whether to sample according to a similarity score with the corpus embeddings.

- **temperature** (`float`) – Sample weight hyper-parameter if `use_proba=True`.
- ****kwargs** – Other arguments. Not used.

Return type`Tuple[ndarray, ndarray]`**Returns**

- *raw* – Array of perturbed text instances.
- *data* – Matrix with 1s and 0s indicating whether a word in the text has not been perturbed for each sample.

set_data_type()

Working with *numpy* arrays of strings requires setting the data type to avoid truncating examples. This function estimates the longest sentence expected during the sampling process, which is used to set the number of characters for the samples and examples arrays. This depends on the perturbation method used for sampling.

Return type`None`**set_text(text)**

Sets the text to be processed

Parameters

text (`str`) – Text to be processed.

Return type`None`

class `alibi.explainers.anchors.text_samplers.UnknownSampler(nlp, perturb_opts)`

Bases: `AnchorTextSampler`

UNK: `str` = 'UNK'

Unknown token to be used.

__call__(*anchor*, *num_samples*)

The function returns a *numpy* array of *num_samples* where randomly chosen features, except those in *anchor*, are replaced by 'UNK' token.

Parameters

- **anchor** (`tuple`) – Indices represent the positions of the words to be kept unchanged.
- **num_samples** (`int`) – Number of perturbed sentences to be returned.

Return type`Tuple[ndarray, ndarray]`**Returns**

- *raw* – Array containing *num_samples* elements. Each element is a perturbed sentence.
- *data* – A (*num_samples*, *m*)-dimensional boolean array, where *m* is the number of tokens in the instance to be explained.

__init__(*nlp*, *perturb_opts*)

Initialize unknown sampler. This sampler replaces word with the *UNK* token.

Parameters

- **nlp** (Language) – *spaCy* object.

- **perturb_opts** (*Dict*) – Perturbation options.

set_data_type()

Working with *numpy* arrays of strings requires setting the data type to avoid truncating examples. This function estimates the longest sentence expected during the sampling process, which is used to set the number of characters for the samples and examples arrays. This depends on the perturbation method used for sampling.

Return type

None

set_text(text)

Sets the text to be processed.

Parameters

text (*str*) – Text to be processed.

Return type

None

alibi.explainers.anchors.text_samplers.load_spacy_lexeme_prob(nlp)

This utility function loads the *lexeme_prob* table for a spacy model if it is not present. This is required to enable support for different spacy versions.

Return type

Language

alibi.explainers.backends package**Subpackages****alibi.explainers.backends.pytorch package****Submodules****alibi.explainers.backends.pytorch.cfrl_base module**

This module contains utility functions for the Counterfactual with Reinforcement Learning base class, [*alibi.explainers.cfrl_base*](#) for the Pytorch backend.

```
class alibi.explainers.backends.pytorch.cfrl_base.PtCounterfactualRLDataset(X, preprocessor,  
                                                                           predictor,  
                                                                           conditional_func,  
                                                                           batch_size)
```

Bases: [*CounterfactualRLDataset*](#), *Dataset*

Pytorch backend datasets.

```
__init__(X, preprocessor, predictor, conditional_func, batch_size)
```

Constructor.

Parameters

- **X** (*ndarray*) – Array of input instances. The input should NOT be preprocessed as it will be preprocessed when calling the *preprocessor* function.

- **preprocessor** (*Callable*) – Preprocessor function. This function correspond to the pre-processing steps applied to the auto-encoder model.
- **predictor** (*Callable*) – Prediction function. The classifier function should expect the input in the original format and preprocess it internally in the *predictor* if necessary.
- **conditional_func** (*Callable*) – Conditional function generator. Given an preprocessed input array, the functions generates a conditional array.
- **batch_size** (*int*) – Dimension of the batch used during training. The same batch size is used to infer the classification labels of the input dataset.

```
alibi.explainers.backends.pytorch.cfml_base.add_noise(Z_cf, noise, act_low, act_high, step,
                                                    exploration_steps, device, **kwargs)
```

Add noise to the counterfactual embedding.

Parameters

- **Z_cf** (*Tensor*) – Counterfactual embedding.
- **noise** (*NormalActionNoise*) – Noise generator object.
- **act_low** (*float*) – Action lower bound.
- **act_high** (*float*) – Action upper bound.
- **step** (*int*) – Training step.
- **exploration_steps** (*int*) – Number of exploration steps. For the first *exploration_steps*, the noised counterfactual embedding is sampled uniformly at random.
- **device** (*device*) – Device to send data to.

Return type

Tensor

Returns

Z_cf_tilde – Noised counterfactual embedding.

```
alibi.explainers.backends.pytorch.cfml_base.consistency_loss(Z_cf_pred, Z_cf_tgt)
```

Default 0 consistency loss.

Parameters

- **Z_cf_pred** (*Tensor*) – Counterfactual embedding prediction.
- **Z_cf_tgt** (*Tensor*) – Counterfactual embedding target.

Returns

0 consistency loss.

```
alibi.explainers.backends.pytorch.cfml_base.data_generator(X, encoder_preprocessor, predictor,
                                                         conditional_func, batch_size, shuffle,
                                                         num_workers, **kwargs)
```

Constructs a tensorflow data generator.

Parameters

- **X** (*ndarray*) – Array of input instances. The input should NOT be preprocessed as it will be preprocessed when calling the *preprocessor* function.
- **encoder_preprocessor** (*Callable*) – Preprocessor function. This function correspond to the preprocessing steps applied to the encoder/auto-encoder model.

- **predictor** (*Callable*) – Prediction function. The classifier function should expect the input in the original format and preprocess it internally in the *predictor* if necessary.
- **conditional_func** (*Callable*) – Conditional function generator. Given an preprocessed input array, the functions generates a conditional array.
- **batch_size** (*int*) – Dimension of the batch used during training. The same batch size is used to infer the classification labels of the input dataset.
- **shuffle** (*bool*) – Whether to shuffle the dataset each epoch. True by default.
- **num_workers** (*int*) – Number of worker processes to be created.
- ****kwargs** – Other arguments. Not used.

`alibi.explainers.backends.pytorch.cfml_base.decode(Z, decoder, device, **kwargs)`

Decodes an embedding tensor.

Parameters

- **Z** (*Tensor*) – Embedding tensor to be decoded.
- **decoder** (*Module*) – Pretrained decoder network.
- **device** (*device*) – Device to sent data to.

Returns

Embedding tensor decoding.

`alibi.explainers.backends.pytorch.cfml_base.encode(X, encoder, device, **kwargs)`

Encodes the input tensor.

Parameters

- **X** (*Tensor*) – Input to be encoded.
- **encoder** (*Module*) – Pretrained encoder network.
- **device** (*device*) – Device to send data to.

Returns

Input encoding.

`alibi.explainers.backends.pytorch.cfml_base.generate_cf(Z, Y_m, Y_t, C, encoder, decoder, actor, device, **kwargs)`

Generates counterfactual embedding.

Parameters

- **Z** (*Tensor*) – Input embedding tensor.
- **Y_m** (*Tensor*) – Input classification label.
- **Y_t** (*Tensor*) – Target counterfactual classification label.
- **C** (*Optional*[*Tensor*]) – Conditional tensor.
- **encoder** (*Module*) – Pretrained encoder network.
- **decoder** (*Module*) – Pretrained decoder network.
- **actor** (*Module*) – Actor network. The model generates the counterfactual embedding.
- **device** (*device*) – Device object to be used.

Return type

Tensor

Returns*Z_{cf}* – Counterfactual embedding.`alibi.explainers.backends.pytorch.cfrl_base.get_actor(hidden_dim, output_dim)`

Constructs the actor network.

Parameters

- **hidden_dim** (`int`) – Actor’s hidden dimension
- **output_dim** (`int`) – Actor’s output dimension.

Return type

Module

Returns*Actor network.*`alibi.explainers.backends.pytorch.cfrl_base.get_critic(hidden_dim)`

Constructs the critic network.

Parameters**hidden_dim** (`int`) – Critic’s hidden dimension.**Return type**

Module

Returns*Critic network.*`alibi.explainers.backends.pytorch.cfrl_base.get_device()`Checks if *cuda* is available. If available, use *cuda* by default, else use *cpu*.**Return type**

device

Returns*Device to be used.*`alibi.explainers.backends.pytorch.cfrl_base.get_optimizer(model, lr=0.001)`Constructs default *Adam* optimizer.**Return type**

Optimizer

Returns*Default optimizer.*`alibi.explainers.backends.pytorch.cfrl_base.load_model(path)`

Loads a model and its optimizer.

Parameters**path** (`Union[str, PathLike]`) – Path to the loading location.**Return type**

Module

Returns*Loaded model.*`alibi.explainers.backends.pytorch.cfrl_base.save_model(path, model)`

Saves a model and its optimizer.

Parameters

- **path** (`Union[str, PathLike]`) – Path to the saving location.
- **model** (`Module`) – Model to be saved.

Return type`None`

```
alibi.explainers.backends.pytorch.cfrl_base.set_seed(seed=13)
```

Sets a seed to ensure reproducibility.

Parameters

seed (`int`) – Seed to be set.

```
alibi.explainers.backends.pytorch.cfrl_base.sparsity_loss(X_hat_cf, X)
```

Default L1 sparsity loss.

Parameters

- **X_hat_cf** (`Tensor`) – Auto-encoder counterfactual reconstruction.
- **X** (`Tensor`) – Input instance

Return type`Dict[str, Tensor]`**Returns**

L1 sparsity loss.

```
alibi.explainers.backends.pytorch.cfrl_base.to_numpy(X)
```

Converts given tensor to *numpy* array.

Parameters

X (`Union[List, ndarray, Tensor, None]`) – Input tensor to be converted to *numpy* array.

Return type`Union[List, ndarray, None]`**Returns**

Numpy representation of the input tensor.

```
alibi.explainers.backends.pytorch.cfrl_base.to_tensor(X, device, **kwargs)
```

Converts tensor to *torch.Tensor*

Return type`Optional[Tensor]`**Returns**

torch.Tensor conversion.

```
alibi.explainers.backends.pytorch.cfrl_base.update_actor_critic(encoder, decoder, critic, actor,
                                                                optimizer_critic,
                                                                optimizer_actor, sparsity_loss,
                                                                consistency_loss, coeff_sparsity,
                                                                coeff_consistency, X, X_cf, Z,
                                                                Z_cf_tilde, Y_m, Y_t, C, R_tilde,
                                                                device, **kwargs)
```

Training step. Updates actor and critic networks including additional losses.

Parameters

- **encoder** (`Module`) – Pretrained encoder network.
- **decoder** (`Module`) – Pretrained decoder network.

- **critic** (Module) – Critic network.
- **actor** (Module) – Actor network.
- **optimizer_critic** (Optimizer) – Critic’s optimizer.
- **optimizer_actor** (Optimizer) – Actor’s optimizer.
- **sparsity_loss** (Callable) – Sparsity loss function.
- **consistency_loss** (Callable) – Consistency loss function.
- **coeff_sparsity** (float) – Sparsity loss coefficient.
- **coeff_consistency** (float) – Consistency loss coefficient
- **X** (ndarray) – Input array.
- **X_cf** (ndarray) – Counterfactual array.
- **Z** (ndarray) – Input embedding.
- **Z_cf_tilde** (ndarray) – Noised counterfactual embedding.
- **Y_m** (ndarray) – Input classification label.
- **Y_t** (ndarray) – Target counterfactual classification label.
- **C** (Optional[ndarray]) – Conditional tensor.
- **R_tilde** (ndarray) – Noised counterfactual reward.
- **device** (device) – Torch device object.
- ****kwargs** – Other arguments. Not used.

Returns

Dictionary of losses.

alibi.explainers.backends.pytorch.cfml_tabular module

This module contains utility functions for the Counterfactual with Reinforcement Learning tabular class, [alibi.explainers.cfml_tabular](#), for the Pytorch backend.

`alibi.explainers.backends.pytorch.cfml_tabular.consistency_loss(Z_cf_pred, Z_cf_tgt, **kwargs)`

Computes heterogeneous consistency loss.

Parameters

- **Z_cf_pred** (Tensor) – Predicted counterfactual embedding.
- **Z_cf_tgt** (Tensor) – Counterfactual embedding target.

Returns

Heterogeneous consistency loss.

`alibi.explainers.backends.pytorch.cfml_tabular.l0_ohe(input, target, reduction='none')`

Computes the L0 loss for a one-hot encoding representation.

Parameters

- **input** (Tensor) – Input tensor.
- **target** (Tensor) – Target tensor
- **reduction** (str) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'.

Return type

Tensor

Returns*L0 loss.*

```
alibi.explainers.backends.pytorch.cfml_tabular.l1_loss(input, target, reduction='none')
```

Computes L1 loss.

Parameters

- **input** (Tensor) – Input tensor.
- **target** (Tensor) – Target tensor.
- **reduction** (str) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'.

Return type

Tensor

Returns*L1 loss.*

```
alibi.explainers.backends.pytorch.cfml_tabular.sample_differentiable(X_hat_split,  
                                                                    category_map)
```

Samples differentiable reconstruction.

Parameters

- **X_hat_split** (List[Tensor]) – List of reconstructed columns from the auto-encoder.
- **category_map** (Dict[int, List[str]]) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible values for an attribute.

Return type

List[Tensor]

Returns*Differentiable reconstruction.*

```
alibi.explainers.backends.pytorch.cfml_tabular.sparsity_loss(X_hat_split, X_ohe, category_map,  
                                                            weight_num=1.0, weight_cat=1.0)
```

Computes heterogeneous sparsity loss.

Parameters

- **X_hat_split** (List[Tensor]) – List of one-hot encoded reconstructed columns from the auto-encoder.
- **X_ohe** (Tensor) – One-hot encoded representation of the input.
- **category_map** (Dict[int, List[str]]) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible values for an attribute.
- **weight_num** (float) – Numerical loss weight.
- **weight_cat** (float) – Categorical loss weight.

Returns*Heterogeneous sparsity loss.*

alibi.explainers.backends.tensorflow package

Submodules

alibi.explainers.backends.tensorflow.cfml_base module

This module contains utility functions for the Counterfactual with Reinforcement Learning base class, `alibi.explainers.cfml_base`, for the Tensorflow backend.

```
class alibi.explainers.backends.tensorflow.cfml_base.TfCounterfactualRLDataset(X, preprocessor,
                                                                              predictor,
                                                                              conditional_func,
                                                                              batch_size,
                                                                              shuffle=True)
```

Bases: `CounterfactualRLDataset`, `Sequence`

Tensorflow backend datasets.

```
__init__(X, preprocessor, predictor, conditional_func, batch_size, shuffle=True)
```

Constructor.

Parameters

- **X** (`ndarray`) – Array of input instances. The input should NOT be preprocessed as it will be preprocessed when calling the `preprocessor` function.
- **preprocessor** (`Callable`) – Preprocessor function. This function correspond to the pre-processing steps applied to the encoder/auto-encoder model.
- **predictor** (`Callable`) – Prediction function. The classifier function should expect the input in the original format and preprocess it internally in the `predictor` if necessary.
- **conditional_func** (`Callable`) – Conditional function generator. Given an pre-processed input array, the functions generates a conditional array.
- **batch_size** (`int`) – Dimension of the batch used during training. The same batch size is used to infer the classification labels of the input dataset.
- **shuffle** (`bool`) – Whether to shuffle the dataset each epoch. True by default.

```
on_epoch_end()
```

This method is called every epoch and performs dataset shuffling.

Return type

`None`

```
alibi.explainers.backends.tensorflow.cfml_base.add_noise(Z_cf, noise, act_low, act_high, step,
                                                         exploration_steps, **kwargs)
```

Add noise to the counterfactual embedding.

Parameters

- **Z_cf** (`Union[Tensor, ndarray]`) – Counterfactual embedding.
- **noise** (`NormalActionNoise`) – Noise generator object.
- **act_low** (`float`) – Noise lower bound.

- **act_high** (*float*) – Noise upper bound.
- **step** (*int*) – Training step.
- **exploration_steps** (*int*) – Number of exploration steps. For the first *exploration_steps*, the noised counterfactual embedding is sampled uniformly at random.
- ****kwargs** – Other arguments. Not used.

Return type

Tensor

Returns

Z_cf_tilde – Noised counterfactual embedding.

`alibi.explainers.backends.tensorflow.cfml_base.consistency_loss(Z_cf_pred, Z_cf_tgt)`

Default 0 consistency loss.

Parameters

- **Z_cf_pred** (Tensor) – Counterfactual embedding prediction.
- **Z_cf_tgt** (Tensor) – Counterfactual embedding target.

Returns

0 consistency loss.

`alibi.explainers.backends.tensorflow.cfml_base.data_generator(X, encoder_preprocessor, predictor, conditional_func, batch_size, shuffle=True, **kwargs)`

Constructs a *tensorflow* data generator.

Parameters

- **X** (ndarray) – Array of input instances. The input should NOT be preprocessed as it will be preprocessed when calling the *preprocessor* function.
- **encoder_preprocessor** (Callable) – Preprocessor function. This function correspond to the preprocessing steps applied to the encoder/auto-encoder model.
- **predictor** (Callable) – Prediction function. The classifier function should expect the input in the original format and preprocess it internally in the *predictor* if necessary.
- **conditional_func** (Callable) – Conditional function generator. Given an preprocessed input array, the functions generates a conditional array.
- **batch_size** (*int*) – Dimension of the batch used during training. The same batch size is used to infer the classification labels of the input dataset.
- **shuffle** (*bool*) – Whether to shuffle the dataset each epoch. True by default.
- ****kwargs** – Other arguments. Not used.

`alibi.explainers.backends.tensorflow.cfml_base.decode(Z, decoder, **kwargs)`

Decodes an embedding tensor.

Parameters

- **Z** (*Union*[Tensor, ndarray]) – Embedding tensor to be decoded.
- **decoder** (Model) – Pretrained decoder network.
- ****kwargs** – Other arguments. Not used.

Returns*Embedding tensor decoding.*`alibi.explainers.backends.tensorflow.cfml_base.encode(X, encoder, **kwargs)`

Encodes the input tensor.

Parameters

- **X** (`Union`[Tensor, ndarray]) – Input to be encoded.
- **encoder** (Model) – Pretrained encoder network.
- ****kwargs** – Other arguments. Not used.

Return type

Tensor

Returns*Input encoding.*`alibi.explainers.backends.tensorflow.cfml_base.generate_cf(Z, Y_m, Y_t, C, actor, **kwargs)`

Generates counterfactual embedding.

Parameters

- **Z** (`Union`[ndarray, Tensor]) – Input embedding tensor.
- **Y_m** (`Union`[ndarray, Tensor]) – Input classification label.
- **Y_t** (`Union`[ndarray, Tensor]) – Target counterfactual classification label.
- **C** (`Union`[ndarray, Tensor, None]) – Conditional tensor.
- **actor** (Model) – Actor network. The model generates the counterfactual embedding.
- ****kwargs** – Other arguments. Not used.

Return type

Tensor

Returns*Z_{cf} – Counterfactual embedding.*`alibi.explainers.backends.tensorflow.cfml_base.get_actor(hidden_dim, output_dim)`

Constructs the actor network.

Parameters

- **hidden_dim** (`int`) – Actor's hidden dimension
- **output_dim** (`int`) – Actor's output dimension.

Return type

Layer

Returns*Actor network.*`alibi.explainers.backends.tensorflow.cfml_base.get_critic(hidden_dim)`

Constructs the critic network.

Parameters**hidden_dim** (`int`) – Critic's hidden dimension.**Return type**

Layer

Returns

Critic network.

```
alibi.explainers.backends.tensorflow.cfml_base.get_optimizer(model=None, lr=0.001)
```

Constructs default *Adam* optimizer.

Parameters

- **model** (`Optional[Layer]`) – Model to get the optimizer for. Not required for *tensorflow* backend.
- **lr** (`float`) – Learning rate.

Return type

Optimizer

Returns

Default optimizer.

```
alibi.explainers.backends.tensorflow.cfml_base.initialize_actor_critic(actor, critic, Z,
                                                                    Z_cf_tilde, Y_m, Y_t, C,
                                                                    **kwargs)
```

Initialize actor and critic layers by passing a dummy zero tensor.

Parameters

- **actor** – Actor model.
- **critic** – Critic model.
- **Z** – Input embedding.
- **Z_cf_tilde** – Noised counterfactual embedding.
- **Y_m** – Input classification label.
- **Y_t** – Target counterfactual classification label.
- **C** – Conditional tensor.
- ****kwargs** – Other arguments. Not used.

```
alibi.explainers.backends.tensorflow.cfml_base.initialize_optimizer(optimizer, model)
```

Initializes an optimizer given a model.

Parameters

- **optimizer** (`Optimizer`) – Optimizer to be initialized.
- **model** (`Model`) – Model to be optimized

Return type

`None`

```
alibi.explainers.backends.tensorflow.cfml_base.initialize_optimizers(optimizer_actor,
                                                                    optimizer_critic, actor,
                                                                    critic, **kwargs)
```

Initializes the actor and critic optimizers.

Parameters

- **optimizer_actor** – Actor optimizer to be initialized.
- **optimizer_critic** – Critic optimizer to be initialized.
- **actor** – Actor model to be optimized.

- **critic** – Critic model to be optimized.
- ****kwargs** – Other arguments. Not used.

Return type*None*`alibi.explainers.backends.tensorflow.cfml_base.load_model(path)`

Loads a model and its optimizer.

Parameters**path** (`Union[str, PathLike]`) – Path to the loading location.**Return type***Model***Returns***Loaded model.*`alibi.explainers.backends.tensorflow.cfml_base.save_model(path, model)`

Saves a model and its optimizer.

Parameters

- **path** (`Union[str, PathLike]`) – Path to the saving location.
- **model** (*Layer*) – Model to be saved.

Return type*None*`alibi.explainers.backends.tensorflow.cfml_base.set_seed(seed=13)`

Sets a seed to ensure reproducibility. Does NOT ensure reproducibility.

Parameters**seed** (`int`) – seed to be set`alibi.explainers.backends.tensorflow.cfml_base.sparsity_loss(X_hat_cf, X)`

Default L1 sparsity loss.

Parameters

- **X_hat_cf** (*Tensor*) – Auto-encoder counterfactual reconstruction.
- **X** (*Tensor*) – Input instance.

Return type`Dict[str, Tensor]`**Returns***L1 sparsity loss.*`alibi.explainers.backends.tensorflow.cfml_base.to_numpy(X)`Converts given tensor to *numpy* array.**Parameters****X** (`Union[List, ndarray, Tensor, None]`) – Input tensor to be converted to *numpy* array.**Return type**`Union[List, ndarray, None]`**Returns***Numpy* representation of the input tensor.

`alibi.explainers.backends.tensorflow.cfml_base.to_tensor(X, **kwargs)`

Converts tensor to *tf.Tensor*.

Parameters

- **X** (`Union[ndarray, Tensor]`) – Input array/tensor to be converted.
- ****kwargs** – Other arguments. Not used.

Return type

`Optional[Tensor]`

Returns

tf.Tensor conversion.

`alibi.explainers.backends.tensorflow.cfml_base.update_actor_critic(encoder, decoder, critic, actor, optimizer_critic, optimizer_actor, sparsity_loss, consistency_loss, coeff_sparsity, coeff_consistency, X, X_cf, Z, Z_cf_tilde, Y_m, Y_t, C, R_tilde, **kwargs)`

Training step. Updates actor and critic networks including additional losses.

Parameters

- **encoder** (`Model`) – Pretrained encoder network.
- **decoder** (`Model`) – Pretrained decoder network.
- **critic** (`Model`) – Critic network.
- **actor** (`Model`) – Actor network.
- **optimizer_critic** (`Optimizer`) – Critic’s optimizer.
- **optimizer_actor** (`Optimizer`) – Actor’s optimizer.
- **sparsity_loss** (`Callable`) – Sparsity loss function.
- **consistency_loss** (`Callable`) – Consistency loss function.
- **coeff_sparsity** (`float`) – Sparsity loss coefficient.
- **coeff_consistency** (`float`) – Consistency loss coefficient
- **X** (`ndarray`) – Input array.
- **X_cf** (`ndarray`) – Counterfactual array.
- **Z** (`ndarray`) – Input embedding.
- **Z_cf_tilde** (`ndarray`) – Noised counterfactual embedding.
- **Y_m** (`ndarray`) – Input classification label.
- **Y_t** (`ndarray`) – Target counterfactual classification label.
- **C** (`Optional[ndarray]`) – Conditional tensor.
- **R_tilde** (`ndarray`) – Noised counterfactual reward.
- ****kwargs** – Other arguments. Not used.

Return type`Dict[str, Any]`**Returns***Dictionary of losses.***alibi.explainers.backends.tensorflow.cfml_tabular module**

This module contains utility functions for the Counterfactual with Reinforcement Learning tabular class (*cfml_tabular*) for the Tensorflow backend.

`alibi.explainers.backends.tensorflow.cfml_tabular.consistency_loss(Z_cf_pred, Z_cf_tgt, **kwargs)`

Computes heterogeneous consistency loss.

Parameters

- **Z_cf_pred** (Tensor) – Counterfactual embedding prediction.
- **Z_cf_tgt** (`Union`[ndarray, Tensor]) – Counterfactual embedding target.

Returns*Heterogeneous consistency loss.*

`alibi.explainers.backends.tensorflow.cfml_tabular.l0_ohc(input, target, reduction='none')`

Computes the L0 loss for a one-hot encoding representation.

Parameters

- **input** (Tensor) – Input tensor.
- **target** (Tensor) – Target tensor
- **reduction** (`str`) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'.

Return type

Tensor

Returns*L0 loss.*

`alibi.explainers.backends.tensorflow.cfml_tabular.l1_loss(input, target=tensorflow.Tensor, reduction='none')`

Computes the L1 loss.

Parameters

- **input** (Tensor) – Input tensor.
- **target** – Target tensor
- **reduction** (`str`) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'.

Return type

Tensor

Returns*L1 loss.*

`alibi.explainers.backends.tensorflow.cfml_tabular.sample_differentiable(X_hat_split, category_map)`

Samples differentiable reconstruction.

Parameters

- **X_hat_split** (`List[Tensor]`) – List of reconstructed columns form the auto-encoder.
- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible values for an attribute.

Return type`List[Tensor]`**Returns***Differentiable reconstruction.*

```
alibi.explainers.backends.tensorflow.cfml_tabular.sparsity_loss(X_hat_split, X_ohc,  
                                                                category_map, weight_num=1.0,  
                                                                weight_cat=1.0)
```

Computes heterogeneous sparsity loss.

Parameters

- **X_hat_split** (`List[Tensor]`) – List of reconstructed columns form the auto-encoder.
- **X_ohc** (`Tensor`) – One-hot encoded representation of the input.
- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible values for an attribute.
- **weight_num** (`float`) – Numerical loss weight.
- **weight_cat** (`float`) – Categorical loss weight.

Returns*Heterogeneous sparsity loss.*

Submodules

alibi.explainers.backends.cfml_base module

This module contains utility functions for the Counterfactual with Reinforcement Learning base class, `alibi.explainers.cfml_base`, that are common for both Tensorflow and Pytorch backends.

```
class alibi.explainers.backends.cfml_base.CounterfactualRLDataset
```

Bases: `ABC`

```
static predict_batches(X, predictor, batch_size)
```

Predict the classification labels of the input dataset. This is performed in batches.

Parameters

- **X** (`ndarray`) – Input to be classified.
- **predictor** (`Callable`) – Prediction function.
- **batch_size** (`int`) – Maximum batch size to be used during each inference step.

Return type`ndarray`**Returns***Classification labels.*

`alibi.explainers.backends.cfrl_base.generate_empty_condition(X)`

Empty conditioning.

Parameters

X (*Any*) – Input instance.

Return type

None

`alibi.explainers.backends.cfrl_base.get_classification_reward(Y_pred, Y_true)`

Computes classification reward per instance given the prediction output and the true label. The classification reward is a sparse/binary reward: 1 if the most likely classes from the prediction output and the label match, 0 otherwise.

Parameters

- **Y_pred** (ndarray) – Prediction output as a distribution over the possible classes.
- **Y_true** (ndarray) – True label as a distribution over the possible classes.

Returns

Classification reward per instance. 1 if the most likely classes match, 0 otherwise.

`alibi.explainers.backends.cfrl_base.get_hard_distribution(Y, num_classes=None)`

Constructs the hard label distribution (one-hot encoding).

Parameters

- **Y** (ndarray) – Prediction array. Can be soft or hard label distribution, or a label.
- **num_classes** (*Optional[int]*) – Number of classes to be considered.

Return type

ndarray

Returns

Hard label distribution (one-hot encoding).

`alibi.explainers.backends.cfrl_base.identity_function(X)`

Identity function.

Parameters

X (*Any*) – Input instance.

Return type

Any

Returns

X – The input instance.

alibi.explainers.backends.cfrl_tabular module

This module contains utility functions for the Counterfactual with Reinforcement Learning tabular class, [alibi.explainers.cfrl_tabular](#), that are common for both Tensorflow and Pytorch backends.

`alibi.explainers.backends.cfrl_tabular.apply_category_mapping(X, category_map)`

Applies a category mapping for the categorical feature in the array. It transforms ints back to strings to be readable.

Parameters

- **X** (ndarray) – Array containing the columns to be mapped.

- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. Keys are columns index, and values are list of feature values.

Return type
ndarray

Returns
Transformed array.

```
alibi.explainers.backends.cfrl_tabular.generate_categorical_condition(X_ohe, feature_names,  
                                                                    category_map,  
                                                                    immutable_features,  
                                                                    conditional=True)
```

Generates categorical features conditional vector. For a categorical feature of cardinality K , we condition the subset of allowed feature through a binary mask of dimension K . When training the counterfactual generator, the mask values are sampled from $Bern(0.5)$. For immutable features, only the original input feature value is set to one in the binary mask. For example, the immutability of the 'marital_status' having the current value 'married' is encoded through the binary sequence [1, 0, 0], given an ordering of the possible feature values [married, unmarried, divorced].

Parameters

- **X_ohe** (ndarray) – One-hot encoding representation of the element(s) for which the conditional vector will be generated. The elements are required since some features can be immutable. In that case, the mask vector is the one-hot encoding itself for that particular feature.
- **feature_names** (`List[str]`) – List of feature names. This should be provided by the dataset.
- **category_map** (`Dict[int, List]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values.
- **immutable_features** (`List[str]`) – List of immutable features.
- **conditional** (bool) – Boolean flag to generate a conditional vector. If False the conditional vector does not impose any restrictions on the feature value.

Return type
ndarray

Returns
Conditional vector for categorical feature.

```
alibi.explainers.backends.cfrl_tabular.generate_condition(X_ohe, feature_names, category_map,  
                                                         ranges, immutable_features,  
                                                         conditional=True)
```

Generates conditional vector.

Parameters

- **X_ohe** (ndarray) – One-hot encoding representation of the element(s) for which the conditional vector will be generated. This method assumes that the input array, *X_ohe*, has the first columns corresponding to the numerical features, and the rest are one-hot encodings of the categorical columns. The numerical and the categorical columns are ordered by the original column index (e.g., *numerical = (1, 4)*, *categorical=(0, 2, 3)*).
- **feature_names** (`List[str]`) – List of feature names.
- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values.

- **ranges** (`Dict[str, List[float]]`) – Dictionary of ranges for numerical features. Each value is a list containing two elements, first one negative and the second one positive.
- **immutable_features** (`List[str]`) – List of immutable map features.
- **conditional** (`bool`) – Boolean flag to generate a conditional vector. If `False` the conditional vector does not impose any restrictions on the feature value.

Return type

ndarray

Returns*Conditional vector.*

```
alibi.explainers.backends.cfml_tabular.generate_numerical_condition(X_ohe, feature_names,
                                                                    category_map, ranges,
                                                                    immutable_features,
                                                                    conditional=True)
```

Generates numerical features conditional vector. For numerical features with a minimum value a_{min} and a maximum value a_{max} , we include in the conditional vector the values $-p_{min}, p_{max}$, where p_{min}, p_{max} are in $[0, 1]$. The range $[-p_{min}, p_{max}]$ encodes a shift and scale-invariant representation of the interval $[a - p_{min} * (a_{max} - a_{min}), a + p_{max} * (a_{max} - a_{min})]$, where a is the original feature value. During training, p_{min} and p_{max} are sampled from $Beta(2, 2)$ for each unconstrained feature. Immutable features can be encoded by $p_{min} = p_{max} = 0$ or listed in `immutable_features` list. Features allowed to increase or decrease only correspond to setting $p_{min} = 0$ or $p_{max} = 0$, respectively. For example, allowing the 'Age' feature to increase by up to 5 years is encoded by taking $p_{min} = 0, p_{max} = 0.1$, assuming the minimum age of 10 and the maximum age of 60 years in the training set: $5 = 0.1 * (60 - 10)$.

Parameters

- **X_ohe** (ndarray) – One-hot encoding representation of the element(s) for which the conditional vector will be generated. This argument is used to extract the number of conditional vector. The choice of `X_ohe` instead of a `size` argument is for consistency purposes with `categorical_cond` function.
- **feature_names** (`List[str]`) – List of feature names. This should be provided by the dataset.
- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values.
- **ranges** (`Dict[str, List[float]]`) – Dictionary of ranges for numerical features. Each value is a list containing two elements, first one negative and the second one positive.
- **immutable_features** (`List[str]`) – Dictionary of immutable features. The keys are the column indexes and the values are booleans: `True` if the feature is immutable, `False` otherwise.
- **conditional** (`bool`) – Boolean flag to generate a conditional vector. If `False` the conditional vector does not impose any restrictions on the feature value.

Return type

ndarray

Returns*Conditional vector for numerical features.*

```
alibi.explainers.backends.cfrl_tabular.get_categorical_conditional_vector(X, condition,
                                                                    preprocessor,
                                                                    feature_names,
                                                                    category_map, im-
                                                                    mutable_features=None,
                                                                    diverse=False)
```

Generates a conditional vector. The condition is expressed as a delta change of the feature. For categorical feature, if the 'Occupation' can change to 'Blue-Collar' or 'White-Collar', the delta change is ['Blue-Collar', 'White-Collar']. Note that the original value is optional as it is included by default.

Parameters

- **X** (ndarray) – Instances for which to generate the conditional vector in the original input format.
- **condition** (Dict[str, List[Union[float, str]]]) – Dictionary of conditions per feature. For numerical features it expects a range that contains the original value. For categorical features it expects a list of feature values per features that includes the original value.
- **preprocessor** (Callable[[ndarray], ndarray]) – Data preprocessor. The preprocessor should standardize the numerical values and convert categorical ones into one-hot encoding representation. By convention, numerical features should be first, followed by the rest of categorical ones.
- **feature_names** (List[str]) – List of feature names. This should be provided by the dataset.
- **category_map** (Dict[int, List[str]]) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values. This should be provided by the dataset.
- **immutable_features** (Optional[List[str]]) – List of immutable features.
- **diverse** – Whether to generate a diverse set of conditional vectors. A diverse set of conditional vector can generate a diverse set of counterfactuals for a given input instance.

Return type

List[ndarray]

Returns

List of conditional vectors for each categorical feature.

```
alibi.explainers.backends.cfrl_tabular.get_conditional_dim(feature_names, category_map)
```

Computes the dimension of the conditional vector.

Parameters

- **feature_names** (List[str]) – List of feature names. This should be provided by the dataset.
- **category_map** (Dict[int, List[str]]) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values. This should be provided by the dataset.

Return type

int

Returns

Dimension of the conditional vector

```
alibi.explainers.backends.cfrl_tabular.get_conditional_vector(X, condition, preprocessor,
                                                            feature_names, category_map,
                                                            stats, ranges=None,
                                                            immutable_features=None,
                                                            diverse=False)
```

Generates a conditional vector. The condition is expressed as a delta change of the feature.

For numerical features, if the 'Age' feature is allowed to increase up to 10 more years, the delta change is [0, 10]. If the 'Hours per week' is allowed to decrease down to -5 and increases up to +10, then the delta change is [-5, +10]. Note that the interval must go include 0.

For categorical feature, if the 'Occupation' can change to 'Blue-Collar' or 'White-Collar', the delta change is ['Blue-Collar', 'White-Collar']. Note that the original value is optional as it is included by default.

Parameters

- **X** (ndarray) – Instances for which to generate the conditional vector in the original input format.
- **condition** (Dict[str, List[Union[float, str]]]) – Dictionary of conditions per feature. For numerical features it expects a range that contains the original value. For categorical features it expects a list of feature values per features that includes the original value.
- **preprocessor** (Callable[[ndarray], ndarray]) – Data preprocessor. The preprocessor should standardize the numerical values and convert categorical ones into one-hot encoding representation. By convention, numerical features should be first, followed by the rest of categorical ones.
- **feature_names** (List[str]) – List of feature names. This should be provided by the dataset.
- **category_map** (Dict[int, List[str]]) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values. This should be provided by the dataset.
- **stats** (Dict[int, Dict[str, float]]) – Dictionary of statistic of the training data. Contains the minimum and maximum value of each numerical feature in the training set. Each key is an index of the column and each value is another dictionary containing 'min' and 'max' keys.
- **ranges** (Optional[Dict[str, List[float]]]) – Dictionary of ranges for numerical feature. Each value is a list containing two elements, first one negative and the second one positive.
- **immutable_features** (Optional[List[str]]) – List of immutable features.
- **diverse** – Whether to generate a diverse set of conditional vectors. A diverse set of conditional vector can generate a diverse set of counterfactuals for a given input instance.

Return type

ndarray

Returns

Conditional vector.

```
alibi.explainers.backends.cfrl_tabular.get_he_preprocessor(X, feature_names, category_map,
                                                          feature_types=None)
```

Heterogeneous dataset preprocessor. The numerical features are standardized and the categorical features are one-hot encoded.

Parameters

- **X** (ndarray) – Data to fit.
- **feature_names** (List[str]) – List of feature names. This should be provided by the dataset.
- **category_map** (Dict[int, List[str]]) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values. This should be provided by the dataset.
- **feature_types** (Optional[Dict[str, type]]) – Dictionary of type for the numerical features.

Return type

Tuple[Callable[[ndarray], ndarray], Callable[[ndarray], ndarray]]

Returns

- *preprocessor* – Data preprocessor.
- *inv_preprocessor* – Inverse data preprocessor (e.g., *inv_preprocessor(preprocessor(x)) = x*)

```
alibi.explainers.backends.cfrl_tabular.get_numerical_conditional_vector(X, condition,
                                                                    preprocessor,
                                                                    feature_names,
                                                                    category_map, stats,
                                                                    ranges=None, im-
                                                                    mutable_features=None,
                                                                    diverse=False)
```

Generates a conditional vector. The condition is expressed as a delta change of the feature. For numerical features, if the 'Age' feature is allowed to increase up to 10 more years, the delta change is [0, 10]. If the 'Hours per week' is allowed to decrease down to -5 and increases up to +10, then the delta change is [-5, +10]. Note that the interval must include 0.

Parameters

- **X** (ndarray) – Instances for which to generate the conditional vector in the original input format.
- **condition** (Dict[str, List[Union[float, str]]) – Dictionary of conditions per feature. For numerical features it expects a range that contains the original value. For categorical features it expects a list of feature values per features that includes the original value.
- **preprocessor** (Callable[[ndarray], ndarray]) – Data preprocessor. The preprocessor should standardize the numerical values and convert categorical ones into one-hot encoding representation. By convention, numerical features should be first, followed by the rest of categorical ones.
- **feature_names** (List[str]) – List of feature names. This should be provided by the dataset.
- **category_map** (Dict[int, List[str]]) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values. This should be provided by the dataset.
- **stats** (Dict[int, Dict[str, float]]) – Dictionary of statistic of the training data. Contains the minimum and maximum value of each numerical feature in the training set. Each key is an index of the column and each value is another dictionary containing 'min' and 'max' keys.

- **ranges** (`Optional[Dict[str, List[float]]]`) – Dictionary of ranges for numerical feature. Each value is a list containing two elements, first one negative and the second one positive.
- **immutable_features** (`Optional[List[str]]`) – List of immutable features.
- **diverse** – Whether to generate a diverse set of conditional vectors. A diverse set of conditional vector can generate a diverse set of counterfactuals for a given input instance.

Return type`List[ndarray]`**Returns***List of conditional vectors for each numerical feature.*`alibi.explainers.backends.cfml_tabular.get_statistics(X, preprocessor, category_map)`

Computes statistics.

Parameters

- **X** (`ndarray`) – Instances for which to compute statistic.
- **preprocessor** (`Callable[[ndarray], ndarray]`) – Data preprocessor. The preprocessor should standardize the numerical values and convert categorical ones into one-hot encoding representation. By convention, numerical features should be first, followed by the rest of categorical ones.
- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values. This should be provided by the dataset.

Return type`Dict[int, Dict[str, float]]`**Returns***Dictionary of statistics. For each numerical column, the minimum and maximum value is returned.*`alibi.explainers.backends.cfml_tabular.sample(X_hat_split, X_ohe, C, category_map, stats)`

Samples an instance from the given reconstruction according to the conditional vector and the dictionary of statistics.

Parameters

- **X_hat_split** (`List[ndarray]`) – List of reconstructed columns from the auto-encoder. The categorical columns contain logits.
- **X_ohe** (`ndarray`) – One-hot encoded representation of the input.
- **C** (`Optional[ndarray]`) – Conditional vector.
- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible values for a feature.
- **stats** (`Dict[int, Dict[str, float]]`) – Dictionary of statistic of the training data. Contains the minimum and maximum value of each numerical feature in the training set. Each key is an index of the column and each value is another dictionary containing 'min' and 'max' keys.

Return type`List[ndarray]`

Returns

X_oh_hat_split – Most probable reconstruction sample according to the auto-encoder, sampled according to the conditional vector and the dictionary of statistics. This method assumes that the input array, *X_oh*, has the first columns corresponding to the numerical features, and the rest are one-hot encodings of the categorical columns.

`alibi.explainers.backends.cfml_tabular.sample_categorical(X_hat_cat_split, C_cat_split)`

Samples categorical features according to the conditional vector. This method sample conditional according to the masking vector the most probable outcome.

Parameters

- **X_hat_cat_split** (`List[ndarray]`) – List of reconstructed categorical heads from the auto-encoder. The categorical columns contain logits.
- **C_cat_split** (`Optional[List[ndarray]]`) – List of conditional vector for categorical heads.

Return type

`List[ndarray]`

Returns

X_oh_hat_cat – List of one-hot encoded vectors sampled according to the conditional vector.

`alibi.explainers.backends.cfml_tabular.sample_numerical(X_hat_num_split, X_oh_num_split, C_num_split, stats)`

Samples numerical features according to the conditional vector. This method clips the values between the desired ranges specified in the conditional vector, and ensures that the values are between the minimum and the maximum values from train training datasets stored in the dictionary of statistics.

Parameters

- **X_hat_num_split** (`List[ndarray]`) – List of reconstructed numerical heads from the auto-encoder. This list should contain a single element as all the numerical features are part of a single linear layer output.
- **X_oh_num_split** (`List[ndarray]`) – List of original numerical heads. The list should contain a single element as part of the convention mentioned in the description of *X_oh_hat_num*.
- **C_num_split** (`Optional[List[ndarray]]`) – List of conditional vector for numerical heads. The list should contain a single element as part of the convention mentioned in the description of *X_oh_hat_num*.
- **stats** (`Dict[int, Dict[str, float]]`) – Dictionary of statistic of the training data. Contains the minimum and maximum value of each numerical feature in the training set. Each key is an index of the column and each value is another dictionary containing 'min' and 'max' keys.

Return type

`List[ndarray]`

Returns

X_oh_hat_num – List of clamped input vectors according to the conditional vectors and the dictionary of statistics.

`alibi.explainers.backends.cfml_tabular.split_oh(X_oh, category_map)`

Splits a one-hot encoding array in a list of numerical heads and a list of categorical heads. Since by convention the numerical heads are merged in a single head, if the function returns a list of numerical heads, then the size of the list is 1.

Parameters

- **X_ohe** (`Union[ndarray, Tensor, Tensor]`) – One-hot encoding representation. This can be any type of tensor: *np.ndarray*, *torch.Tensor*, *tf.Tensor*.
- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible values of a feature.

Return type`Tuple[List, List]`**Returns**

- *X_ohe_num_split* – List of numerical heads. If different than None, the list's size is 1.
- *X_ohe_cat_split* – List of categorical one-hot encoded heads.

alibi.explainers.similarity package**Subpackages****alibi.explainers.similarity.backends package****Subpackages****alibi.explainers.similarity.backends.pytorch package****Submodules****alibi.explainers.similarity.backends.pytorch.base module**

pytorch backend for similarity explainers.

Methods unique to the *pytorch* backend are defined here. The interface this class defines syncs with the *tensorflow* backend in order to ensure that the similarity methods only require to match this interface.

alibi.explainers.similarity.backends.tensorflow package**Submodules****alibi.explainers.similarity.backends.tensorflow.base module**

tensorflow backend for similarity explainers.

Methods unique to the *tensorflow* backend are defined here. The interface this class defines syncs with the *pytorch* backend in order to ensure that the similarity methods only require to match this interface.

Submodules

alibi.explainers.similarity.base module

```
class alibi.explainers.similarity.base.BaseSimilarityExplainer(predictor, loss_fn, sim_fn,
                                                                precompute_grads=False, back-
                                                                end=Framework.TENSORFLOW,
                                                                device=None, meta=None,
                                                                verbose=False)
```

Bases: [Explainer](#), [ABC](#)

Base class for similarity explainers.

```
__init__(predictor, loss_fn, sim_fn, precompute_grads=False, backend=Framework.TENSORFLOW,
          device=None, meta=None, verbose=False)
```

Constructor

Parameters

- **predictor** ([Union](#)[[Model](#), [Module](#)]) – Model to be explained.
- **loss_fn** ([Union](#)[[Callable](#)[[[Tensor](#), [Tensor](#)], [Tensor](#)], [Callable](#)[[[Tensor](#), [Tensor](#)], [Tensor](#)]]) – Loss function.
- **sim_fn** ([Callable](#)[[[ndarray](#), [ndarray](#)], [ndarray](#)]) – Similarity function. Takes two inputs and returns a similarity value.
- **precompute_grads** ([bool](#)) – Whether to precompute and store the gradients when fitting.
- **backend** ([Framework](#)) – Deep learning backend.
- **device** ([Union](#)[[int](#), [str](#), [device](#), [None](#)]) – Device to be used. Will default to the same device the backend defaults to.
- **meta** ([Optional](#)[[dict](#)]) – Metadata specific to explainers that inherit from this class. Should be initialized in the child class and passed in here. Is used in the `__init__` of the base [Explainer](#) class.

```
fit(X_train, Y_train)
```

Fit the explainer. If `self.precompute_grads == True` then the gradients are precomputed and stored.

Parameters

- **X_train** ([Union](#)[[ndarray](#), [List](#)[[Any](#)]]) – Training data.
- **Y_train** ([ndarray](#)) – Training labels.

Return type

[Explainer](#)

Returns

self – Returns self.

```
reset_predictor(predictor)
```

Resets the predictor to the given predictor.

Parameters

- **predictor** ([Union](#)[[Model](#), [Module](#)]) – The new predictor to use.

Return type

[None](#)

alibi.explainers.similarity.grad module

Gradient-based explainer.

This module implements the gradient-based explainers grad-dot and grad-cos.

```
class alibi.explainers.similarity.grad.GradientSimilarity(predictor, loss_fn, sim_fn='grad_dot',
                                                         task='classification',
                                                         precompute_grads=False,
                                                         backend='tensorflow', device=None,
                                                         verbose=False)
```

Bases: [BaseSimilarityExplainer](#)

```
__init__(predictor, loss_fn, sim_fn='grad_dot', task='classification', precompute_grads=False,
         backend='tensorflow', device=None, verbose=False)
```

GradientSimilarity explainer.

The gradient similarity explainer is used to find examples in the training data that the predictor considers similar to test instances the user wants to explain. It uses the gradients of the loss between the model output and the training data labels. These are compared using the similarity function specified by `sim_fn`. The *GradientSimilarity* explainer can be applied to models trained for both classification and regression tasks.

Parameters

- **predictor** ([Union](#)[[Model](#), [Module](#)]) – Model to explain.
- **loss_fn** ([Union](#)[[Callable](#)[[[Tensor](#), [Tensor](#)], [Tensor](#)], [Callable](#)[[[Tensor](#), [Tensor](#)], [Tensor](#)]]) – Loss function used. The gradient of the loss function is used to compute the similarity between the test instances and the training set.
- **sim_fn** ([Literal](#)['grad_dot', 'grad_cos', 'grad_asym_dot']) – Similarity function to use. The 'grad_dot' similarity function computes the dot product of the gradients, see [alibi.explainers.similarity.metrics.dot\(\)](#). The 'grad_cos' similarity function computes the cosine similarity between the gradients, see [alibi.explainers.similarity.metrics.cos\(\)](#). The 'grad_asym_dot' similarity function is similar to 'grad_dot' but is asymmetric, see [alibi.explainers.similarity.metrics.asym_dot\(\)](#).
- **task** ([Literal](#)['classification', 'regression']) – Type of task performed by the model. If the task is 'classification', the target value passed to the explain method of the test instance can be specified either directly or left as `None`, if left `None` we use the model's maximum prediction. If the task is 'regression', the target value of the test instance must be specified directly.
- **precompute_grads** ([bool](#)) – Whether to precompute the gradients. If `False`, gradients are computed on the fly otherwise we precompute them which can be faster when it comes to computing explanations. Note this option may be memory intensive if the model is large.
- **backend** ([Literal](#)['tensorflow', 'pytorch']) – Backend to use.
- **device** ([Union](#)[[int](#), [str](#), [device](#), [None](#)]) – Device to use. If `None`, the default device for the backend is used. If using *pytorch* backend see [pytorch device docs](#) for correct options. Note that in the *pytorch* backend case this parameter can be a `torch.device`. If using *tensorflow* backend see [tensorflow docs](#) for correct options.
- **verbose** ([bool](#)) – Whether to print the progress of the explainer.

Raises

- **ValueError** – If the task is not 'classification' or 'regression'.
- **ValueError** – If the `sim_fn` is not 'grad_dot', 'grad_cos' or 'grad_asym_dot'.

- **ValueError** – If the backend is not 'tensorflow' or 'pytorch'.
- **TypeError** – If the device is not an int, str, torch.device or None for the torch backend option or if the device is not str or None for the tensorflow backend option.

explain(X, Y=None)

Explain the predictor's predictions for a given input.

Computes the similarity score between the inputs and the training set. Returns an explainer object containing the scores, the indices of the training set instances sorted by descending similarity and the most similar and least similar instances of the data set for the input. Note that the input may be a single instance or a batch of instances.

Parameters

- **X** (`Union[ndarray, Tensor, Tensor, Any, List[Any]]`) – X can be a *numpy* array, *tensorflow* tensor, *pytorch* tensor of the same shape as the training data or a list of objects, with or without a leading batch dimension. If the batch dimension is missing it's added.
- **Y** (`Union[ndarray, Tensor, Tensor, None]`) – Y can be a *numpy* array, *tensorflow* tensor or a *pytorch* tensor. In the case of a regression task, the Y argument must be present. If the task is classification then Y defaults to the model prediction.

Return type

Explanation

Returns

Explanation object containing the ordered similarity scores for the test instance(s) with additional metadata as attributes. Contains the following data-related attributes –

- **scores**: `np.ndarray` - similarity scores for each pair of instances in the training and test set sorted in descending order.
- **ordered_indices**: `np.ndarray` - indices of the paired training and test set instances sorted by the similarity score in descending order.
- **most_similar**: `np.ndarray` - 5 most similar instances in the training set for each test instance. The first element is the most similar instance.
- **least_similar**: `np.ndarray` - 5 least similar instances in the training set for each test instance. The first element is the least similar instance.

Raises

- **ValueError** – If Y is None and the *task* is 'regression'.
- **ValueError** – If the shape of X or Y does not match the shape of the training or target data.
- **ValueError** – If the fit method has not been called prior to calling this method.

fit(X_train, Y_train)

Fit the explainer.

The *GradientSimilarity* explainer requires the model gradients over the training data. In the explain method it compares them to the model gradients for the test instance(s). If `precompute_grads=True` on initialization then the gradients are precomputed here and stored. This will speed up the explain method call but storing the gradients may not be feasible for large models.

Parameters

- **X_train** (`Union[ndarray, List[Any]]`) – Training data.

- **Y_train** (ndarray) – Training labels.

Return type*Explainer***Returns***self* – Returns self.

```
class alibi.explainers.similarity.grad.Task(value)
```

Bases: `str`, `Enum`

Enum of supported tasks.

CLASSIFICATION = 'classification'

REGRESSION = 'regression'

alibi.explainers.similarity.metrics module

```
alibi.explainers.similarity.metrics.asym_dot(X, Y, eps=1e-07)
```

Computes the influence of training instances Y to test instances X . This is an asymmetric kernel. $(X^T Y / \|Y\|^2)$. See the [paper](#) for more details. Each of X and Y should have a leading batch dimension of size at least 1.

Parameters

- **X** (ndarray) – Matrix of vectors.
- **Y** (ndarray) – Matrix of vectors.
- **eps** (float) – Numerical stability.

Return type`Union[float, ndarray]`**Returns**

Matrix of asymmetric dot product similarity values between the vector(s) in X and vectors in Y .

```
alibi.explainers.similarity.metrics.cos(X, Y, eps=1e-07)
```

Computes the cosine between the vector(s) in X and vector Y . $(X^T Y / \|X\| \|Y\|)$. Each of X and Y should have a leading batch dimension of size at least 1.

Parameters

- **X** (ndarray) – Matrix of vectors.
- **Y** (ndarray) – Matrix of vectors.
- **eps** (float) – Numerical stability.

Return type`Union[float, ndarray]`**Returns**

Matrix of cosine similarities between the vector(s) in X and vectors in Y .

```
alibi.explainers.similarity.metrics.dot(X, Y)
```

Performs a dot product between the vector(s) in X and vector Y . $(X^T Y = \sum_i X_i Y_i)$. Each of X and Y should have a leading batch dimension of size at least 1.

Parameters

- **X** (ndarray) – Matrix of vectors.

- **Y** (ndarray) – Matrix of vectors.

Return type

`Union[float, ndarray]`

Returns

Matrix of dot products between the vector(s) in X and vectors in Y.

Submodules

alibi.explainers.ale module

```
class alibi.explainers.ale.ALE(predictor, feature_names=None, target_names=None,
                               check_feature_resolution=True, low_resolution_threshold=10,
                               extrapolate_constant=True, extrapolate_constant_perc=10.0,
                               extrapolate_constant_min=0.1)
```

Bases: [Explainer](#)

```
__init__(predictor, feature_names=None, target_names=None, check_feature_resolution=True,
          low_resolution_threshold=10, extrapolate_constant=True, extrapolate_constant_perc=10.0,
          extrapolate_constant_min=0.1)
```

Accumulated Local Effects for tabular datasets. Current implementation supports first order feature effects of numerical features.

Parameters

- **predictor** (`Callable[[ndarray], ndarray]`) – A callable that takes in an $N \times F$ array as input and outputs an $N \times T$ array (N - number of data points, F - number of features, T - number of outputs/targets (e.g. 1 for single output regression, ≥ 2 for classification)).
- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.
- **target_names** (`Optional[List[str]]`) – A list of target/output names used for displaying results.
- **check_feature_resolution** (`bool`) – If `True`, the number of unique values is calculated for each feature and if it is less than `low_resolution_threshold` then the feature values are used for grid-points instead of quantiles. This may increase the runtime of the algorithm for large datasets. Only used for features without custom grid-points specified in [alibi.explainers.ale.ALE.explain\(\)](#).
- **low_resolution_threshold** (`int`) – If a feature has at most this many unique values, these are used as the grid points instead of quantiles. This is to avoid situations when the quantile algorithm returns quantiles between discrete values which can result in jumps in the ALE plot obscuring the true effect. Only used if `check_feature_resolution` is `True` and for features without custom grid-points specified in [alibi.explainers.ale.ALE.explain\(\)](#).
- **extrapolate_constant** (`bool`) – If a feature is constant, only one quantile exists where all the data points lie. In this case the ALE value at that point is zero, however this may be misleading if the feature does have an effect on the model. If this parameter is set to `True`, the ALE values are calculated on an interval surrounding the constant value. The interval length is controlled by the `extrapolate_constant_perc` and `extrapolate_constant_min` arguments.
- **extrapolate_constant_perc** (`float`) – Percentage by which to extrapolate a constant feature value to create an interval for ALE calculation. If q is the constant feature value,

creates an interval $[q - q/\text{extrapolate_constant_perc}, q + q/\text{extrapolate_constant_perc}]$ for which ALE is calculated. Only relevant if `extrapolate_constant` is set to `True`.

- **extrapolate_constant_min** (`float`) – Controls the minimum extrapolation length for constant features. An interval constructed for constant features is guaranteed to be $2 \times \text{extrapolate_constant_min}$ wide centered on the feature value. This allows for capturing model behaviour around constant features which have small value so that `extrapolate_constant_perc` is not so helpful. Only relevant if `extrapolate_constant` is set to `True`.

explain(*X*, *features=None*, *min_bin_points=4*, *grid_points=None*)

Calculate the ALE curves for each feature with respect to the dataset *X*.

Parameters

- **X** (`ndarray`) – An $N \times F$ tabular dataset used to calculate the ALE curves. This is typically the training dataset or a representative sample.
- **features** (`Optional[List[int]]`) – Features for which to calculate ALE.
- **min_bin_points** (`int`) – Minimum number of points each discretized interval should contain to ensure more precise ALE estimation. Only relevant for adaptive grid points (i.e., features without an entry in the `grid_points` dictionary).
- **grid_points** (`Optional[Dict[int, ndarray]]`) – Custom grid points. Must be a *dict* where the keys are features indices and the values are monotonically increasing *numpy* arrays defining the grid points for each feature. See the *Notes* section for the default behavior when potential edge-cases arise when using grid-points. If no grid points are specified (i.e. the feature is missing from the `grid_points` dictionary), deciles discretization is used instead.

Return type

Explanation

Returns

explanation – An *Explanation* object containing the data and the metadata of the calculated ALE curves. See usage at [ALE examples](#) for details.

Notes

Consider *f* to be a feature of interest. We denote possible feature values of *f* by *X* (i.e. the values from the dataset column corresponding to feature *f*), by *O* a user-specified grid-point value, and by $(X|O)$ an overlap between a grid-point and a feature value. We can encounter the following edge-cases:

- Grid points outside the feature range. Consider the following example: *O O O X X O X O X O O*, where 3 grid-points are smaller than the minimum value in *f*, and 2 grid-points are larger than the maximum value in *f*. The empty leading and ending bins are removed. The grid-points considered

will be: *O X X O X O X O*.

- Grid points that do not cover the entire feature range. Consider the following example: *X X O X X O X O X X X X X*. Two auxiliary grid-points are added which correspond the value of the minimum and maximum value of feature *f*. The grid-points considered will be: $(O|X) X O X X O X O X X X X (X|O)$.
- Grid points that do not contain any values in between. Consider the following example: $(O|X) X X O O O X O X O O (X|O)$. The intervals which do not contain any feature values are removed/merged. The grid-points considered will be: $(O|X) X X O X O X O (X|O)$.

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor ([Callable](#)) – New predictor function.

Return type

[None](#)

`alibi.explainers.ale.adaptive_grid(values, min_bin_points=1)`

Find the optimal number of quantiles for the range of values so that each resulting bin contains at least *min_bin_points*. Uses bisection.

Parameters

- **values** ([ndarray](#)) – Array of feature values.
- **min_bin_points** ([int](#)) – Minimum number of points each discretized interval should contain to ensure more precise ALE estimation.

Return type

[Tuple](#)[[ndarray](#), [int](#)]

Returns

- *q* – Unique quantiles.
- *num_quantiles* – Number of non-unique quantiles the feature array was subdivided into.

Notes

This is a heuristic procedure since the bisection algorithm is applied to a function which is not monotonic. This will not necessarily find the maximum number of bins the interval can be subdivided into to satisfy the minimum number of points in each resulting bin.

`alibi.explainers.ale.ale_num(predictor, X, feature, feature_grid_points=None, min_bin_points=4, check_feature_resolution=True, low_resolution_threshold=10, extrapolate_constant=True, extrapolate_constant_perc=10.0, extrapolate_constant_min=0.1)`

Calculate the first order ALE curve for a numerical feature.

Parameters

- **predictor** ([Callable](#)) – Model prediction function.
- **X** ([ndarray](#)) – Dataset for which ALE curves are computed.
- **feature** ([int](#)) – Index of the numerical feature for which to calculate ALE.
- **feature_grid_points** ([Optional](#)[[ndarray](#)]) – Custom grid points. An *numpy* array defining the grid points for the given features.
- **min_bin_points** ([int](#)) – Minimum number of points each discretized interval should contain to ensure more precise ALE estimation. Only relevant for adaptive grid points (i.e., feature for which *feature_grid_points=None*).
- **check_feature_resolution** ([bool](#)) – Refer to [ALE](#) documentation.
- **low_resolution_threshold** ([int](#)) – Refer to [ALE](#) documentation.
- **extrapolate_constant** ([bool](#)) – Refer to [ALE](#) documentation.
- **extrapolate_constant_perc** ([float](#)) – Refer to [ALE](#) documentation.
- **extrapolate_constant_min** ([float](#)) – Refer to [ALE](#) documentation.

Return type`Tuple[ndarray, ...]`**Returns**

- *fvals* – Array of quantiles or custom grid-points of the input values.
- *ale* – ALE values for each feature at each of the points in *fvals*.
- *ale0* – The constant offset used to center the ALE curves.

`alibi.explainers.ale.bisect_fun(fun, target, lo, hi)`

Bisection algorithm for function evaluation with integer support.

Assumes the function is non-decreasing on the interval $[lo, hi]$. Return an integer value v such that for all $x < v$, $fun(x) < target$ and for all $x \geq v$, $fun(x) \geq target$. This is equivalent to the library function `bisect.bisect_left` but for functions defined on integers.

Parameters

- **fun** (`Callable`) – A function defined on integers in the range $[lo, hi]$ and returning floats.
- **target** (`float`) – Target value to be searched for.
- **lo** (`int`) – Lower bound of the domain.
- **hi** (`int`) – Upper bound of the domain.

Return type`int`**Returns**

Integer index.

`alibi.explainers.ale.get_quantiles(values, num_quantiles=11, interpolation='linear')`

Calculate quantiles of values in an array.

Parameters

- **values** (`ndarray`) – Array of values.
- **num_quantiles** (`int`) – Number of quantiles to calculate.

Return type`ndarray`**Returns**

Array of quantiles of the input values.

`alibi.explainers.ale.minimum_satisfied(values, min_bin_points, n)`

Calculates whether the partition into bins induced by n quantiles has the minimum number of points in each resulting bin.

Parameters

- **values** (`ndarray`) – Array of feature values.
- **min_bin_points** (`int`) – Minimum number of points each discretized interval needs to contain.
- **n** (`int`) – Number of quantiles.

Return type`int`

Returns

Integer encoded boolean with 1 - each bin has at least *min_bin_points* and 0 otherwise.

`alibi.explainers.ale.plot_ale(exp, features='all', targets='all', n_cols=3, sharey='all', constant=False, ax=None, line_kw=None, fig_kw=None)`

Plot ALE curves on matplotlib axes.

Parameters

- **exp** – An *Explanation* object produced by a call to the `alibi.explainers.ale.ALE.explain()` method.
- **features** – A list of features for which to plot the ALE curves or 'all' for all features. Can be a mix of integers denoting feature index or strings denoting entries in *exp.feature_names*. Defaults to 'all'.
- **targets** – A list of targets for which to plot the ALE curves or 'all' for all targets. Can be a mix of integers denoting target index or strings denoting entries in *exp.target_names*. Defaults to 'all'.
- **n_cols** – Number of columns to organize the resulting plot into.
- **sharey** – A parameter specifying whether the y-axis of the ALE curves should be on the same scale for several features. Possible values are: 'all' | 'row' | None.
- **constant** – A parameter specifying whether the constant zeroth order effects should be added to the ALE first order effects.
- **ax** – A *matplotlib* axes object or a *numpy* array of *matplotlib* axes to plot on.
- **line_kw** – Keyword arguments passed to the *plt.plot* function.
- **fig_kw** – Keyword arguments passed to the *fig.set* function.

Returns

An array of *matplotlib* axes with the resulting ALE plots.

alibi.explainers.cem module

```
class alibi.explainers.cem.CEM(predict, mode, shape, kappa=0.0, beta=0.1,
                               feature_range=(-10000000000.0, 10000000000.0), gamma=0.0,
                               ae_model=None, learning_rate_init=0.01, max_iterations=1000,
                               c_init=10.0, c_steps=10, eps=(0.001, 0.001), clip=(-100.0, 100.0),
                               update_num_grad=1, no_info_val=None, write_dir=None, sess=None)
```

Bases: `Explainer`, `FitMixin`

```
__init__(predict, mode, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0),
          gamma=0.0, ae_model=None, learning_rate_init=0.01, max_iterations=1000, c_init=10.0,
          c_steps=10, eps=(0.001, 0.001), clip=(-100.0, 100.0), update_num_grad=1, no_info_val=None,
          write_dir=None, sess=None)
```

Initialize contrastive explanation method. Paper: <https://arxiv.org/abs/1802.07623>

Parameters

- **predict** (`Union[Callable[[ndarray], ndarray], Model]`) – *tensorflow* model or any other model's prediction function returning class probabilities.
- **mode** (`str`) – Find pertinent negatives (PN) or pertinent positives (PP).
- **shape** (`tuple`) – Shape of input data starting with batch size.

- **kappa** (`float`) – Confidence parameter for the attack loss term.
- **beta** (`float`) – Regularization constant for L1 loss term.
- **feature_range** (`tuple`) – Tuple with min and max ranges to allow for perturbed instances. Min and max ranges can be *float* or *numpy* arrays with dimension (1x nb of features) for feature-wise ranges.
- **gamma** (`float`) – Regularization constant for optional auto-encoder loss term.
- **ae_model** (`Optional[Model]`) – Optional auto-encoder model used for loss regularization.
- **learning_rate_init** (`float`) – Initial learning rate of optimizer.
- **max_iterations** (`int`) – Maximum number of iterations for finding a PN or PP.
- **c_init** (`float`) – Initial value to scale the attack loss term.
- **c_steps** (`int`) – Number of iterations to adjust the constant scaling the attack loss term.
- **eps** (`tuple`) – If numerical gradients are used to compute $dL/dx = (dL/dp) * (dp/dx)$, then *eps[0]* is used to calculate dL/dp and *eps[1]* is used for dp/dx . *eps[0]* and *eps[1]* can be a combination of *float* values and *numpy* arrays. For *eps[0]*, the array dimension should be (1x nb of prediction categories) and for *eps[1]* it should be (1x nb of features).
- **clip** (`tuple`) – Tuple with *min* and *max* clip ranges for both the numerical gradients and the gradients obtained from the *tensorflow* graph.
- **update_num_grad** (`int`) – If numerical gradients are used, they will be updated every *update_num_grad* iterations.
- **no_info_val** (`Union[float, ndarray, None]`) – Global or feature-wise value considered as containing no information.
- **write_dir** (`Optional[str]`) – Directory to write *tensorboard* files to.
- **sess** (`Optional[Session]`) – Optional *tensorflow* session that will be used if passed instead of creating or inferring one internally.

attack(*X*, *Y*, *verbose=False*)

Find pertinent negative or pertinent positive for instance *X* using a fast iterative shrinkage-thresholding algorithm (FISTA).

Parameters

- **X** (`ndarray`) – Instance to attack.
- **Y** (`ndarray`) – Labels for *X*.
- **verbose** (`bool`) – Print intermediate results of optimization if `True`.

Return type

`Tuple[ndarray, Tuple[ndarray, ndarray]]`

Returns

Overall best attack and gradients for that attack.

explain(*X*, *Y=None*, *verbose=False*)

Explain instance and return PP or PN with metadata.

Parameters

- **X** (`ndarray`) – Instances to attack.
- **Y** (`Optional[ndarray]`) – Labels for *X*.

- **verbose** (`bool`) – Print intermediate results of optimization if `True`.

Return type*Explanation***Returns**

explanation – *Explanation* object containing the PP or PN with additional metadata as attributes. See usage at [CEM examples](#) for details.

fit(*train_data*, *no_info_type*='median')

Get 'no information' values from the training data.

Parameters

- **train_data** (`ndarray`) – Representative sample from the training data.
- **no_info_type** (`str`) – Median or mean value by feature supported.

Return type*CEM*

get_gradients(*X*, *Y*)

Compute numerical gradients of the attack loss term: $dL/dx = (dL/dP)*(dP/dx)$ with $L = loss_attack_s$; $P = predict$; $x = adv_s$

Parameters

- **X** (`ndarray`) – Instance around which gradient is evaluated.
- **Y** (`ndarray`) – One-hot representation of instance labels.

Return type`ndarray`**Returns***Array with gradients.*

loss_fn(*pred_proba*, *Y*)

Compute the attack loss.

Parameters

- **pred_proba** (`ndarray`) – Prediction probabilities of an instance.
- **Y** (`ndarray`) – One-hot representation of instance labels.

Return type`ndarray`**Returns***Loss of the attack.*

perturb(*X*, *eps*, *proba*=`False`)

Apply perturbation to instance or prediction probabilities. Used for numerical calculation of gradients.

Parameters

- **X** (`ndarray`) – Array to be perturbed.
- **eps** (`Union[float, ndarray]`) – Size of perturbation.
- **proba** (`bool`) – If `True`, the net effect of the perturbation needs to be 0 to keep the sum of the probabilities equal to 1.

Return type`Tuple[ndarray, ndarray]`**Returns***Instances where a positive and negative perturbation is applied.***reset_predictor**(*predictor*)

Resets the predictor function/model.

Parameters**predictor** (`Union[Callable, Model]`) – New predictor function/model.**Return type**`None`**alibi.explainers.cfproto module**`alibi.explainers.cfproto.CounterFactualProto(*args, **kwargs)`The class name *CounterFactualProto* is deprecated, please use *CounterfactualProto*.

```
class alibi.explainers.cfproto.CounterfactualProto(predict, shape, kappa=0.0, beta=0.1,
                                                feature_range=(-10000000000.0,
                                                                10000000000.0), gamma=0.0, ae_model=None,
                                                enc_model=None, theta=0.0, cat_vars=None,
                                                ohe=False, use_kdtree=False,
                                                learning_rate_init=0.01, max_iterations=1000,
                                                c_init=10.0, c_steps=10, eps=(0.001, 0.001),
                                                clip=(-1000.0, 1000.0), update_num_grad=1,
                                                write_dir=None, sess=None)
```

Bases: [Explainer](#), [FitMixin](#)

```
__init__(predict, shape, kappa=0.0, beta=0.1, feature_range=(-10000000000.0, 10000000000.0),
          gamma=0.0, ae_model=None, enc_model=None, theta=0.0, cat_vars=None, ohe=False,
          use_kdtree=False, learning_rate_init=0.01, max_iterations=1000, c_init=10.0, c_steps=10,
          eps=(0.001, 0.001), clip=(-1000.0, 1000.0), update_num_grad=1, write_dir=None, sess=None)
```

Initialize prototypical counterfactual method.

Parameters

- **predict** (`Union[Callable[[ndarray], ndarray], Model]`) – *tensorflow* model or any other model's prediction function returning class probabilities.
- **shape** (`tuple`) – Shape of input data starting with batch size.
- **kappa** (`float`) – Confidence parameter for the attack loss term.
- **beta** (`float`) – Regularization constant for L1 loss term.
- **feature_range** (`Tuple[Union[float, ndarray], Union[float, ndarray]]`) – Tuple with *min* and *max* ranges to allow for perturbed instances. *Min* and *max* ranges can be *float* or *numpy* arrays with dimension (1x nb of features) for feature-wise ranges.
- **gamma** (`float`) – Regularization constant for optional auto-encoder loss term.
- **ae_model** (`Optional[Model]`) – Optional auto-encoder model used for loss regularization.
- **enc_model** (`Optional[Model]`) – Optional encoder model used to guide instance perturbations towards a class prototype.

- **theta** (`float`) – Constant for the prototype search loss term.
- **cat_vars** (`Optional[Dict[int, int]]`) – Dict with as keys the categorical columns and as values the number of categories per categorical variable.
- **ohe** (`bool`) – Whether the categorical variables are one-hot encoded (OHE) or not. If not OHE, they are assumed to have ordinal encodings.
- **use_kdtree** (`bool`) – Whether to use k-d trees for the prototype loss term if no encoder is available.
- **learning_rate_init** (`float`) – Initial learning rate of optimizer.
- **max_iterations** (`int`) – Maximum number of iterations for finding a counterfactual.
- **c_init** (`float`) – Initial value to scale the attack loss term.
- **c_steps** (`int`) – Number of iterations to adjust the constant scaling the attack loss term.
- **eps** (`tuple`) – If numerical gradients are used to compute $dL/dx = (dL/dp) * (dp/dx)$, then `eps[0]` is used to calculate dL/dp and `eps[1]` is used for dp/dx . `eps[0]` and `eps[1]` can be a combination of `float` values and `numpy` arrays. For `eps[0]`, the array dimension should be (1x nb of prediction categories) and for `eps[1]` it should be (1x nb of features).
- **clip** (`tuple`) – Tuple with min and max clip ranges for both the numerical gradients and the gradients obtained from the *tensorflow* graph.
- **update_num_grad** (`int`) – If numerical gradients are used, they will be updated every `update_num_grad` iterations.
- **write_dir** (`Optional[str]`) – Directory to write *tensorboard* files to.
- **sess** (`Optional[Session]`) – Optional *tensorflow* session that will be used if passed instead of creating or inferring one internally.

attack(*X*, *Y*, *target_class=None*, *k=None*, *k_type='mean'*, *threshold=0.0*, *verbose=False*, *print_every=100*, *log_every=100*)

Find a counterfactual (CF) for instance *X* using a fast iterative shrinkage-thresholding algorithm (FISTA).

Parameters

- **X** (`ndarray`) – Instance to attack.
- **Y** (`ndarray`) – Labels for *X* as one-hot-encoding.
- **target_class** (`Optional[list]`) – List with target classes used to find closest prototype. If `None`, the nearest prototype except for the predict class on the instance is used.
- **k** (`Optional[int]`) – Number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class if an encoder is used and to 1 for k-d trees.
- **k_type** (`str`) – Use either the average encoding of the *k* nearest instances in a class (`k_type='mean'`) or the *k*-nearest encoding in the class (`k_type='point'`) to define the prototype of that class. Only relevant if an encoder is used to define the prototypes.
- **threshold** (`float`) – Threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements.
- **verbose** (`bool`) – Print intermediate results of optimization if `True`.
- **print_every** (`int`) – Print frequency if `verbose` is `True`.

- **log_every** (`int`) – *tensorboard* log frequency if write directory is specified.

Return type

`Tuple`[`ndarray`, `Tuple`[`ndarray`, `ndarray`]]

Returns

Overall best attack and gradients for that attack.

```
explain(X, Y=None, target_class=None, k=None, k_type='mean', threshold=0.0, verbose=False,
        print_every=100, log_every=100)
```

Explain instance and return counterfactual with metadata.

Parameters

- **X** (`ndarray`) – Instances to attack.
- **Y** (`Optional`[`ndarray`]) – Labels for *X* as one-hot-encoding.
- **target_class** (`Optional`[`list`]) – List with target classes used to find closest prototype. If `None`, the nearest prototype except for the predict class on the instance is used.
- **k** (`Optional`[`int`]) – Number of nearest instances used to define the prototype for a class. Defaults to using all instances belonging to the class if an encoder is used and to 1 for k-d trees.
- **k_type** (`str`) – Use either the average encoding of the *k* nearest instances in a class (`k_type='mean'`) or the k-nearest encoding in the class (`k_type='point'`) to define the prototype of that class. Only relevant if an encoder is used to define the prototypes.
- **threshold** (`float`) – Threshold level for the ratio between the distance of the counterfactual to the prototype of the predicted class for the original instance over the distance to the prototype of the predicted class for the counterfactual. If the trust score is below the threshold, the proposed counterfactual does not meet the requirements.
- **verbose** (`bool`) – Print intermediate results of optimization if `True`.
- **print_every** (`int`) – Print frequency if verbose is `True`.
- **log_every** (`int`) – *tensorboard* log frequency if write directory is specified

Return type

Explanation

Returns

explanation – *Explanation* object containing the counterfactual with additional metadata as attributes. See usage at [CFProto examples](#) for details.

```
fit(train_data, trustscore_kwargs=None, d_type='abdm', w=None, disc_perc=(25, 50, 75),
    standardize_cat_vars=False, smooth=1.0, center=True, update_feature_range=True)
```

Get prototypes for each class using the encoder or k-d trees. The prototypes are used for the encoder loss term or to calculate the optional trust scores.

Parameters

- **train_data** (`ndarray`) – Representative sample from the training data.
- **trustscore_kwargs** (`Optional`[`dict`]) – Optional arguments to initialize the trust scores method.
- **d_type** (`str`) – Pairwise distance metric used for categorical variables. Currently, 'abdm', 'mvdm' and 'abdm-mvdm' are supported. 'abdm' infers context from the other variables while 'mvdm' uses the model predictions. 'abdm-mvdm' is a weighted combination of the two metrics.

- **w** (`Optional[float]`) – Weight on 'abdm' (between 0. and 1.) distance if *d_type* equals 'abdm-mvdm'.
- **disc_perc** (`Sequence[Union[int, float]]`) – List with percentiles used in binning of numerical features used for the 'abdm' and 'abdm-mvdm' pairwise distance measures.
- **standardize_cat_vars** (`bool`) – Standardize numerical values of categorical variables if True.
- **smooth** (`float`) – Smoothing exponent between 0 and 1 for the distances. Lower values will smooth the difference in distance metric between different features.
- **center** (`bool`) – Whether to center the scaled distance measures. If False, the min distance for each feature except for the feature with the highest raw max distance will be the lower bound of the feature range, but the upper bound will be below the max feature range.
- **update_feature_range** (`bool`) – Update feature range with scaled values.

Return type*CounterfactualProto***get_gradients**(*X, Y, grads_shape, cat_vars_ord*)

Compute numerical gradients of the attack loss term: $dL/dx = (dL/dP)*(dP/dx)$ with $L = loss_attack_s$; $P = predict$; $x = adv_s$.

Parameters

- **X** (`ndarray`) – Instance around which gradient is evaluated.
- **Y** (`ndarray`) – One-hot representation of instance labels.
- **grads_shape** (`tuple`) – Shape of gradients.
- **cat_vars_ord** (`dict`) – Dict with as keys the categorical columns and as values the number of categories per categorical variable.

Return type`ndarray`**Returns***Array with gradients.***loss_fn**(*pred_proba, Y*)

Compute the attack loss.

Parameters

- **pred_proba** (`ndarray`) – Prediction probabilities of an instance.
- **Y** (`ndarray`) – One-hot representation of instance labels.

Return type`ndarray`**Returns***Loss of the attack.***reset_predictor**(*predictor*)

Resets the predictor function/model.

Parameters

predictor (`Union[Callable, Model]`) – New predictor function/model.

Return type`None`

score(*X*, *adv_class*, *orig_class*, *eps*=1e-10)

Parameters

- **X** (ndarray) – Instance to encode and calculate distance metrics for.
- **adv_class** (int) – Predicted class on the perturbed instance.
- **orig_class** (int) – Predicted class on the original instance.
- **eps** (float) – Small number to avoid dividing by 0.

Return type

float

Returns

Ratio between the distance to the prototype of the predicted class for the original instance and the prototype of the predicted class for the perturbed instance.

alibi.explainers.cfml_base module

class alibi.explainers.cfml_base.Callback

Bases: ABC

Training callback class.

abstract `__call__`(*step*, *update*, *model*, *sample*, *losses*)

Training callback applied after every training step.

Parameters

- **step** (int) – Current experience step.
- **update** (int) – Current update step. The ration between the number experience steps and the number of training updates is bound to 1.
- **model** (*CounterfactualRL*) – CounterfactualRL explainer. All the parameters defined in `alibi.explainers.cfml_base.DEFAULT_BASE_PARAMS` can be accessed through `model.params`.
- **sample** (Dict[str, ndarray]) – Dictionary of samples used for an update which contains
 - 'X' : np.ndarray - input instances.
 - 'Y_m' : np.ndarray - predictor outputs for the input instances.
 - 'Y_t' : np.ndarray - target outputs.
 - 'Z' : np.ndarray - input embeddings.
 - 'Z_cf_tilde' : np.ndarray - noised counterfactual embeddings.
 - 'X_cf_tilde' : np.ndarray - noised counterfactual instances obtained after decoding the noised counterfactual embeddings `Z_cf_tilde` and apply post-processing functions.
 - 'C' : Optional[np.ndarray] - conditional vector.
 - 'R_tilde' : np.ndarray - reward obtained for the noised counterfactual instances.
 - 'Z_cf' : np.ndarray - counterfactual embeddings.
 - 'X_cf' : np.ndarray - counterfactual instances obtained after decoding the counterfactual embeddings `Z_cf` and apply post-processing functions.
- **losses** (Dict[str, float]) – Dictionary of losses which contains

- 'loss_actor' : Callable - actor network loss.
- 'loss_critic' : Callable - critic network loss.
- 'sparsity_loss' : Callable - sparsity loss for the [alibi.explainers.cfrl_base.CounterfactualRL](#) class.
- 'sparsity_num_loss' : Callable - numerical features sparsity loss for the [alibi.explainers.cfrl_tabular.CounterfactualRLTabular](#) class.
- 'sparsity_cat_loss' : Callable - categorical features sparsity loss for the [alibi.explainers.cfrl_tabular.CounterfactualRLTabular](#) class.
- 'consistency_loss' : Callable - consistency loss if used.

Return type

None

```
class alibi.explainers.cfrl_base.CounterfactualRL(predictor, encoder, decoder, coeff_sparsity,
                                                  coeff_consistency, latent_dim=None,
                                                  backend='tensorflow', seed=0, **kwargs)
```

Bases: [Explainer](#), [FitMixin](#)

Counterfactual Reinforcement Learning.

```
__init__(predictor, encoder, decoder, coeff_sparsity, coeff_consistency, latent_dim=None,
          backend='tensorflow', seed=0, **kwargs)
```

Constructor.

Parameters

- **predictor** (Callable[[ndarray], ndarray]) – A callable that takes a *numpy* array of N data points as inputs and returns N outputs. For classification task, the second dimension of the output should match the number of classes. Thus, the output can be either a soft label distribution or a hard label distribution (i.e. one-hot encoding) without affecting the performance since *argmax* is applied to the predictor's output.
- **encoder** (Union[Model, Module]) – Pretrained encoder network.
- **decoder** (Union[Model, Module]) – Pretrained decoder network.
- **coeff_sparsity** (float) – Sparsity loss coefficient.
- **coeff_consistency** (float) – Consistency loss coefficient.
- **latent_dim** (Optional[int]) – Auto-encoder latent dimension. Can be omitted if the actor network is user specified.
- **backend** (str) – Deep learning backend: 'tensorflow' | 'pytorch'. Default 'tensorflow'.
- **seed** (int) – Seed for reproducibility. The results are not reproducible for 'tensorflow' backend.
- ****kwargs** – Used to replace any default parameter from [alibi.explainers.cfrl_base.DEFAULT_BASE_PARAMS](#).

```
explain(X, Y_t, C=None, batch_size=100)
```

Explains an input instance

Parameters

- **X** (ndarray) – Instances to be explained.
- **Y_t** (ndarray) – Counterfactual targets.

- **C** (`Optional[ndarray]`) – Conditional vectors. If `None`, it means that no conditioning was used during training (i.e. the `conditional_func` returns `None`).
- **batch_size** (`int`) – Batch size to be used when generating counterfactuals.

Return type*Explanation***Returns**

explanation – *Explanation* object containing the counterfactual with additional metadata as attributes. See usage at [CFRL examples](#) for details.

fit(X)

Fit the model agnostic counterfactual generator.

Parameters

X (`ndarray`) – Training data array.

Return type*Explainer***Returns**

self – The explainer itself.

classmethod load(path, predictor)

Load an explainer from disk.

Parameters

- **path** (`Union[str, PathLike]`) – Path to a directory containing the saved explainer.
- **predictor** (`Any`) – Model or prediction function used to originally initialize the explainer.

Return type*Explainer***Returns**

An explainer instance.

reset_predictor(predictor)

Resets the predictor.

Parameters

predictor (`Any`) – New predictor.

Return type`None`**save(path)**

Save an explainer to disk. Uses the *dill* module.

Parameters

path (`Union[str, PathLike]`) – Path to a directory. A new directory will be created if one does not exist.

Return type`None`

```
alibi.explainers.cfml_base.DEFAULT_BASE_PARAMS = {'act_high': 1.0, 'act_low': -1.0,
'act_noise': 0.1, 'actor': None, 'actor_hidden_dim': 256, 'backend': 'tensorflow',
'batch_size': 100, 'callbacks': [], 'conditional_func': <function
generate_empty_condition>, 'critic': None, 'critic_hidden_dim': 256,
'decoder_inv_preprocessor': <function identity_function>, 'encoder_preprocessor':
<function identity_function>, 'exploration_steps': 100, 'lr_actor': 0.001, 'lr_critic':
0.001, 'num_workers': 4, 'optimizer_actor': None, 'optimizer_critic': None,
'postprocessing_funcs': [], 'replay_buffer_size': 1000, 'reward_func': <function
get_classification_reward>, 'shuffle': True, 'train_steps': 100000, 'update_after': 10,
'update_every': 1}
```

Default Counterfactual with Reinforcement Learning parameters.

- 'act_noise' : float - standard deviation for the normal noise added to the actor for exploration.
- 'act_low' : float - minimum action value. Each action component takes values between *[act_low, act_high]*.
- 'act_high' : float - maximum action value. Each action component takes values between *[act_low, act_high]*.
- 'replay_buffer_size' : int - dimension of the replay buffer in *batch_size* units. The total memory allocated is proportional with the *size x batch_size*.
- 'batch_size' : int - training batch size.
- 'num_workers' : int - number of workers used by the data loader if 'pytorch' backend is selected.
- 'shuffle' : bool - whether to shuffle the datasets every epoch.
- 'exploration_steps' : int - number of exploration steps. For the first *exploration_steps*, the counterfactual embedding coordinates are sampled uniformly at random from the interval *[act_low, act_high]*.
- 'update_every' : int - number of steps that should elapse between gradient updates. Regardless of the waiting steps, the ratio of waiting steps to gradient steps is locked to 1.
- 'update_after' : int - number of steps to wait before start updating the actor and critic. This ensures that the replay buffers is full enough for useful updates.
- 'backend' : str - backend to be used: 'tensorflow' | 'pytorch'. Default 'tensorflow'.
- 'train_steps' : int - number of train steps.
- 'encoder_preprocessor' : Callable - encoder/auto-encoder data preprocessors. Transforms the input data into the format expected by the auto-encoder. By default, the identity function.
- 'decoder_inv_preprocessor' : Callable - decoder/auto-encoder data inverse preprocessor. Transforms data from the auto-encoder output format to the original input format. Before calling the prediction function, the data is inverse preprocessed to match the original input format. By default, the identity function.
- 'reward_func' : Callable - element-wise reward function. By default, considers classification task and checks if the counterfactual prediction label matches the target label. Note that this is element-wise, so a tensor is expected to be returned.
- 'postprocessing_funcs' : List[Postprocessing] - list of post-processing functions. The function are applied in the order, from low to high index. Non-differentiable post-processing can be applied. The function expects as arguments *X_cf* - the counterfactual instance, *X* - the original input instance and *C* - the conditional vector, and returns the post-processed counterfactual instance *X_cf_pp* which is passed as *X_cf* for the following functions. By default, no post-processing is applied (empty list).
- 'conditional_func' : Callable - generates a conditional vector given a pre-processed input instance. By default, the function returns None which is equivalent to no conditioning.

- 'callbacks' : List[Callback] - list of callback functions applied at the end of each training step.
- 'actor' : Optional[Union[tensorflow.keras.Model, torch.nn.Module]] - actor network.
- 'critic' : Optional[Union[tensorflow.keras.Model, torch.nn.Module]] - critic network.
- 'optimizer_actor' : Optional[Union[tensorflow.keras.optimizers.Optimizer, torch.optim.Optimizer]] - actor optimizer.
- 'optimizer_critic' : Optional[Union[tensorflow.keras.optimizer.Optimizer, torch.optim.Optimizer]] - critic optimizer.
- 'lr_actor' : float - actor learning rate.
- 'lr_critic' : float - critic learning rate.
- 'actor_hidden_dim' : int - actor hidden layer dimension.
- 'critic_hidden_dim' : int - critic hidden layer dimension.

class alibi.explainers.cfml_base.**NormalActionNoise**(*mu*, *sigma*)

Bases: [object](#)

Normal noise generator.

__call__(*shape*)

Generates normal noise with the appropriate mean and standard deviation.

Parameters

shape (Tuple[int, ...]) – Shape of the array to be generated

Return type

ndarray

Returns

Normal noise with the appropriate mean, standard deviation and shape.

__init__(*mu*, *sigma*)

Constructor.

Parameters

- **mu** (float) – Mean of the normal noise.
- **sigma** (float) – Standard deviation of the noise.

class alibi.explainers.cfml_base.**Postprocessing**

Bases: [ABC](#)

abstract **__call__**(*X_cf*, *X*, *C*)

Post-processing function

Parameters

- **X_cf** (Any) – Counterfactual instance. The datatype depends on the output of the decoder. For example, for an image dataset, the output is np.ndarray. For a tabular dataset, the output is List[np.ndarray] where each element of the list corresponds to a feature. This corresponds to the decoder's output from the heterogeneous autoencoder (see [alibi.models.tensorflow.autoencoder.HeAE](#) and [alibi.models.pytorch.autoencoder.HeAE](#)).
- **X** (ndarray) – Input instance.
- **C** (Optional[ndarray]) – Conditional vector. If None, it means that no conditioning was used during training (i.e. the *conditional_func* returns None).

Return type*Any***Returns** X_{cf} – Post-processed X_{cf} .**class** alibi.explainers.cfml_base.ReplayBuffer(*size=1000*)Bases: *object*

Circular experience replay buffer for *CounterfactualRL* (DDPG). When the buffer is filled, then the oldest experience is replaced by the new one (FIFO). The experience batch size is kept constant and inferred when the first batch of data is stored. Allowing flexible batch size can generate *tensorflow* warning due to the *tf.function* retracing, which can lead to a drop in performance.

R_tilde: ndarray

Noise counterfactual rewards buffer.

X: ndarray

Inputs buffer.

Y_m: ndarray

Model's prediction buffer.

Y_t: ndarray

Counterfactual targets buffer.

Z: ndarray

Input embedding buffer.

Z_cf_tilde: ndarray

Noised counterfactual embedding buffer.

__init__(*size=1000*)

Constructor.

Parameters

size (*int*) – Dimension of the buffer in batch size. This that the total memory allocated is proportional with the *size x batch_size*, where *batch_size* is inferred from the first array to be stored.

append(*X, Y_m, Y_t, Z, Z_cf_tilde, C, R_tilde, **kwargs*)

Adds experience to the replay buffer. When the buffer is filled, then the oldest experience is replaced by the new one (FIFO).

Parameters

- **X** (*ndarray*) – Input array.
- **Y_m** (*ndarray*) – Model's prediction class of *X*.
- **Y_t** (*ndarray*) – Counterfactual target class.
- **Z** (*ndarray*) – Input's embedding.
- **Z_cf_tilde** (*ndarray*) – Noised counterfactual embedding.
- **C** (*Optional[ndarray]*) – Conditional array.
- **R_tilde** (*ndarray*) – Noised counterfactual reward array.
- ****kwargs** – Other arguments. Not used.

Return type

None

sample()

Sample a batch of experience form the replay buffer.

Return type

Dict[str, Optional[ndarray]]

Returns

A batch experience. For a description of the keys and values returned, see parameter descriptions in `alibi.explainers.cfrl_base.ReplayBuffer.append()` method. The batch size returned is the same as the one passed in the `alibi.explainers.cfrl_base.ReplayBuffer.append()`.

alibi.explainers.cfrl_tabular module**class alibi.explainers.cfrl_tabular.ConcatTabularPostprocessing**

Bases: `Postprocessing`

Tabular feature columns concatenation post-processing.

__call__(X_cf, X, C)

Performs a concatenation of the counterfactual feature columns along the axis 1.

Parameters

- **X_cf** (List[ndarray]) – List of counterfactual feature columns.
- **X** (ndarray) – Input instance. Not used. Included for consistency.
- **C** (Optional[ndarray]) – Conditional vector. Not used. Included for consistency.

Return type

ndarray

Returns

Concatenation of the counterfactual feature columns.

```
class alibi.explainers.cfrl_tabular.CounterfactualRLTabular(predictor, encoder, decoder,
                                                            encoder_preprocessor,
                                                            decoder_inv_preprocessor,
                                                            coeff_sparsity, coeff_consistency,
                                                            feature_names, category_map,
                                                            immutable_features=None,
                                                            ranges=None, weight_num=1.0,
                                                            weight_cat=1.0, latent_dim=None,
                                                            backend='tensorflow', seed=0,
                                                            **kwargs)
```

Bases: `CounterfactualRL`

Counterfactual Reinforcement Learning Tabular.

```
__init__(predictor, encoder, decoder, encoder_preprocessor, decoder_inv_preprocessor, coeff_sparsity,
          coeff_consistency, feature_names, category_map, immutable_features=None, ranges=None,
          weight_num=1.0, weight_cat=1.0, latent_dim=None, backend='tensorflow', seed=0, **kwargs)
```

Constructor.

Parameters

- **predictor** (`Callable[[ndarray], ndarray]`) – A callable that takes a *numpy* array of N data points as inputs and returns N outputs. For classification task, the second dimension of the output should match the number of classes. Thus, the output can be either a soft label distribution or a hard label distribution (i.e. one-hot encoding) without affecting the performance since *argmax* is applied to the predictor's output.
- **encoder** (`Union[Model, Module]`) – Pretrained heterogeneous encoder network.
- **decoder** (`Union[Model, Module]`) – Pretrained heterogeneous decoder network. The output of the decoder must be a list of tensors.
- **encoder_preprocessor** (`Callable`) – Auto-encoder data pre-processor. Depending on the input format, the pre-processor can normalize numerical attributes, transform label encoding to one-hot encoding etc.
- **decoder_inv_preprocessor** (`Callable`) – Auto-encoder data inverse pre-processor. This is the inverse function of the pre-processor. It can denormalize numerical attributes, transform one-hot encoding to label encoding, feature type casting etc.
- **coeff_sparsity** (`float`) – Sparsity loss coefficient.
- **coeff_consistency** (`float`) – Consistency loss coefficient.
- **feature_names** (`List[str]`) – List of feature names. This should be provided by the dataset.
- **category_map** (`Dict[int, List[str]]`) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible values for a feature. This should be provided by the dataset.
- **immutable_features** (`Optional[List[str]]`) – List of immutable features.
- **ranges** (`Optional[Dict[str, Tuple[int, int]]]`) – Numerical feature ranges. Note that exist numerical features such as 'Age', which are allowed to increase only. We denote those by 'inc_feat'. Similarly, there exist features allowed to decrease only. We denote them by 'dec_feat'. Finally, there are some free feature, which we denote by 'free_feat'. With the previous notation, we can define `range = {'inc_feat': [0, 1], 'dec_feat': [-1, 0], 'free_feat': [-1, 1]}`. 'free_feat' can be omitted, as any unspecified feature is considered free. Having the ranges of a feature `{'feat': [a_low, a_high]}`, when sampling is performed the numerical value will be clipped between `[a_low * (max_val - min_val), a_high * (max_val - min_val)]`, where `a_low` and `a_high` are the minimum and maximum values the feature 'feat'. This implies that `a_low` and `a_high` are not restricted to `{-1, 0}` and `{0, 1}`, but can be any float number in-between `[-1, 0]` and `[0, 1]`.
- **weight_num** (`float`) – Numerical loss weight.
- **weight_cat** (`float`) – Categorical loss weight.
- **latent_dim** (`Optional[int]`) – Auto-encoder latent dimension. Can be omitted if the actor network is user specified.
- **backend** (`str`) – Deep learning backend: 'tensorflow' | 'pytorch'. Default 'tensorflow'.
- **seed** (`int`) – Seed for reproducibility. The results are not reproducible for 'tensorflow' backend.
- ****kwargs** – Used to replace any default parameter from `alibi.explainers.cfrl_base.DEFAULT_BASE_PARAMS`.

explain(*X*, *Y_t*, *C=None*, *batch_size=100*, *diversity=False*, *num_samples=1*, *patience=1000*, *tolerance=0.001*)

Computes counterfactuals for the given instances conditioned on the target and the conditional vector.

Parameters

- **X** (ndarray) – Input instances to generate counterfactuals for.
- **Y_t** (ndarray) – Target labels.
- **C** (Optional[List[Dict[str, List[Union[float, str]]]]) – List of conditional dictionaries. If *None*, it means that no conditioning was used during training (i.e. the *conditional_func* returns *None*). If conditioning was used during training but no conditioning is desired for the current input, an empty list is expected.
- **diversity** (bool) – Whether to generate diverse counterfactual set for the given instance. Only supported for a single input instance.
- **num_samples** (int) – Number of diversity samples to be generated. Considered only if *diversity=True*.
- **batch_size** (int) – Batch size to use when generating counterfactuals.
- **patience** (int) – Maximum number of iterations to perform diversity search stops. If -1, the search stops only if the desired number of samples has been found.
- **tolerance** (float) – Tolerance to distinguish two counterfactual instances.

Return type

Explanation

Returns

explanation – *Explanation* object containing the counterfactual with additional metadata as attributes. See usage [CFRL examples](#) for details.

fit(X)

Fit the model agnostic counterfactual generator.

Parameters

X (ndarray) – Training data array.

Return type

Explainer

Returns

self – The explainer itself.

class alibi.explainers.cfml_tabular.SampleTabularPostprocessing(*category_map*, *stats*)

Bases: *Postprocessing*

Tabular sampling post-processing. Given the output of the heterogeneous auto-encoder the post-processing functions samples the output according to the conditional vector. Note that the original input instance is required to perform the conditional sampling.

__call__(*X_cf*, *X*, *C*)

Performs counterfactual conditional sampling according to the conditional vector and the original input.

Parameters

- **X_cf** (List[ndarray]) – Decoder reconstruction of the counterfactual instance. The decoded instance is a list where each element in the list correspond to the reconstruction of a feature.

- **X** (ndarray) – Input instance.
- **C** (Optional[ndarray]) – Conditional vector.

Return type`List[ndarray]`**Returns***Conditional sampled counterfactual instance.*`__init__(category_map, stats)`

Constructor.

Parameters

- **category_map** (Dict[int, List[str]]) – Dictionary of category mapping. The keys are column indexes and the values are lists containing the possible feature values.
- **stats** (Dict[int, Dict[str, float]]) – Dictionary of statistic of the training data. Contains the minimum and maximum value of each numerical feature in the training set. Each key is an index of the column and each value is another dictionary containing 'min' and 'max' keys.

alibi.explainers.counterfactual module`alibi.explainers.counterfactual.CounterFactual(*args, **kwargs)`The class name *CounterFactual* is deprecated, please use *Counterfactual*.

```
class alibi.explainers.counterfactual.Counterfactual(predict_fn, shape, distance_fn='l1',
                                                    target_proba=1.0, target_class='other',
                                                    max_iter=1000, early_stop=50, lam_init=0.1,
                                                    max_lam_steps=10, tol=0.05,
                                                    learning_rate_init=0.1,
                                                    feature_range=(-10000000000.0,
                                                                    10000000000.0), eps=0.01, init='identity',
                                                    decay=True, write_dir=None, debug=False,
                                                    sess=None)
```

Bases: [Explainer](#)

```
__init__(predict_fn, shape, distance_fn='l1', target_proba=1.0, target_class='other', max_iter=1000,
          early_stop=50, lam_init=0.1, max_lam_steps=10, tol=0.05, learning_rate_init=0.1,
          feature_range=(-10000000000.0, 10000000000.0), eps=0.01, init='identity', decay=True,
          write_dir=None, debug=False, sess=None)
```

Initialize counterfactual explanation method based on Wachter et al. (2017)

Parameters

- **predict_fn** (Union[Callable[[ndarray], ndarray], Model]) – *tensorflow* model or any other model's prediction function returning class probabilities.
- **shape** (Tuple[int, ...]) – Shape of input data starting with batch size.
- **distance_fn** (str) – Distance function to use in the loss term.
- **target_proba** (float) – Target probability for the counterfactual to reach.
- **target_class** (Union[str, int]) – Target class for the counterfactual to reach, one of 'other', 'same' or an integer denoting desired class membership for the counterfactual instance.

- **max_iter** (`int`) – Maximum number of iterations to run the gradient descent for (inner loop).
- **early_stop** (`int`) – Number of steps after which to terminate gradient descent if all or none of found instances are solutions.
- **lam_init** (`float`) – Initial regularization constant for the prediction part of the Wachter loss.
- **max_lam_steps** (`int`) – Maximum number of times to adjust the regularization constant (outer loop) before terminating the search.
- **tol** (`float`) – Tolerance for the counterfactual target probability.
- **learning_rate_init** – Initial learning rate for each outer loop of *lambda*.
- **feature_range** (`Union[Tuple, str]`) – Tuple with *min* and *max* ranges to allow for perturbed instances. *Min* and *max* ranges can be *float* or *numpy* arrays with dimension (1 x nb of features) for feature-wise ranges.
- **eps** (`Union[float, ndarray]`) – Gradient step sizes used in calculating numerical gradients, defaults to a single value for all features, but can be passed an array for feature-wise step sizes.
- **init** (`str`) – Initialization method for the search of counterfactuals, currently must be 'identity'.
- **decay** (`bool`) – Flag to decay learning rate to zero for each outer loop over *lambda*.
- **write_dir** (`Optional[str]`) – Directory to write *tensorboard* files to.
- **debug** (`bool`) – Flag to write *tensorboard* summaries for debugging.
- **sess** (`Optional[Session]`) – Optional *tensorflow* session that will be used if passed instead of creating or inferring one internally.

explain(X)

Explain an instance and return the counterfactual with metadata.

Parameters

X (`ndarray`) – Instance to be explained.

Return type

Explanation

Returns

explanation – *Explanation* object containing the counterfactual with additional metadata as attributes. See usage at [Counterfactual examples](#) for details.

fit(X, y)

Fit method - currently unused as the counterfactual search is fully unsupervised.

Parameters

- **X** (`ndarray`) – Not used. Included for consistency.
- **y** (`Optional[ndarray]`) – Not used. Included for consistency.

Return type

Counterfactual

Returns

self – Explainer itself.

reset_predictor(*predictor*)

Resets the predictor function/model.

Parameters

predictor (`Union[Callable, Model]`) – New predictor function/model.

Return type

`None`

alibi.explainers.integrated_gradients module

```
class alibi.explainers.integrated_gradients.IntegratedGradients(model, layer=None,
                                                                target_fn=None,
                                                                method='gausslegendre',
                                                                n_steps=50,
                                                                internal_batch_size=100)
```

Bases: `Explainer`

```
__init__(model, layer=None, target_fn=None, method='gausslegendre', n_steps=50,
          internal_batch_size=100)
```

An implementation of the integrated gradients method for *tensorflow* models.

For details of the method see the original paper: <https://arxiv.org/abs/1703.01365> .

Parameters

- **model** (`Model`) – *tensorflow* model.
- **layer** (`Union[Callable[[Model], Layer], Layer, None]`) – A layer or a function having as parameter the model and returning a layer with respect to which the gradients are calculated. If not provided, the gradients are calculated with respect to the input. To guarantee saving and loading of the explainer, the layer has to be specified as a callable which returns a layer given the model. E.g. `lambda model: model.layers[0].embeddings`.
- **target_fn** (`Optional[Callable]`) – A scalar function that is applied to the predictions of the model. This can be used to specify which scalar output the attributions should be calculated for. This can be particularly useful if the desired output is not known before calling the model (e.g. explaining the *argmax* output for a probabilistic classifier, in this case we could pass `target_fn=partial(np.argmax, axis=1)`).
- **method** (`str`) – Method for the integral approximation. Methods available: "riemann_left", "riemann_right", "riemann_middle", "riemann_trapezoid", "gausslegendre".
- **n_steps** (`int`) – Number of step in the path integral approximation from the baseline to the input instance.
- **internal_batch_size** (`int`) – Batch size for the internal batching.

```
explain(X, forward_kwargs=None, baselines=None, target=None, attribute_to_layer_inputs=False)
```

Calculates the attributions for each input feature or element of layer and returns an `Explanation` object.

Parameters

- **X** (`Union[ndarray, List[ndarray]]`) – Instance for which integrated gradients attribution are computed.
- **forward_kwargs** (`Optional[dict]`) – Input keyword args. If it's not `None`, it must be a dict with *numpy* arrays as values. The first dimension of the arrays must correspond to the

number of examples. It will be repeated for each of *n_steps* along the integrated path. The attributions are not computed with respect to these arguments.

- **baselines** (`Union[int, float, ndarray, List[int], List[float], List[ndarray], None]`) – Baselines (starting point of the path integral) for each instance. If the passed value is an *np.ndarray* must have the same shape as *X*. If not provided, all features values for the baselines are set to 0.
- **target** (`Union[int, list, ndarray, None]`) – Defines which element of the model output is considered to compute the gradients. Target can be a numpy array, a list or a numeric value. Numeric values are only valid if the model's output is a rank-n tensor with $n \leq 2$ (regression and classification models). If a numeric value is passed, the gradients are calculated for the same element of the output for all data points. For regression models whose output is a scalar, target should not be provided. For classification models *target* can be either the true classes or the classes predicted by the model. It must be provided for classification models and regression models whose output is a vector. If the model's output is a rank-n tensor with $n > 2$, the target must be a rank-2 numpy array or a list of lists (a matrix) with dimensions *nb_samples* X (*n*-1) .
- **attribute_to_layer_inputs** (`bool`) – In case of layers gradients, controls whether the gradients are computed for the layer's inputs or outputs. If `True`, gradients are computed for the layer's inputs, if `False` for the layer's outputs.

Return type

Explanation

Returns

explanation – *Explanation* object including *meta* and *data* attributes with integrated gradients attributions for each feature. See usage at [IG examples](#) for details.

reset_predictor(*predictor*)

Resets the predictor model.

Parameters

predictor (`Model`) – New prediction model.

Return type

`None`

class `alibi.explainers.integrated_gradients.LayerState`(*value*)

Bases: `str`, `Enum`

An enumeration.

CALLABLE = 'callable'

NON_SERIALIZABLE = 'non-serializable'

UNSPECIFIED = 'unspecified'

alibi.explainers.partial_dependence module

```
class alibi.explainers.partial_dependence.Kind(value)
```

Bases: `str`, `Enum`

Enumeration of supported kind.

AVERAGE = 'average'

BOTH = 'both'

INDIVIDUAL = 'individual'

```
class alibi.explainers.partial_dependence.PartialDependence(predictor, feature_names=None,
                                                            categorical_names=None,
                                                            target_names=None, verbose=False)
```

Bases: `PartialDependenceBase`

Black-box implementation of partial dependence for tabular datasets. Supports multiple feature interactions.

```
__init__(predictor, feature_names=None, categorical_names=None, target_names=None, verbose=False)
```

Initialize black-box model implementation of partial dependence.

Parameters

- **predictor** (`Callable[[ndarray], ndarray]`) – A prediction function which receives as input a *numpy* array of size $N \times F$ and outputs a *numpy* array of size N (i.e. $(N,)$ or $N \times T$, where N is the number of input instances, F is the number of features and T is the number of targets).
- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.
- **categorical_names** (`Optional[Dict[int, List[str]]]`) – Dictionary where keys are feature columns and values are the categories for the feature. Necessary to identify the categorical features in the dataset. An example for *categorical_names* would be:

```
category_map = {0: ["married", "divorced"], 3: ["high school diploma
→", "master's degree"]}
```

- **target_names** (`Optional[List[str]]`) – A list of target/output names used for displaying results.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

Notes

The length of the *target_names* should match the number of columns returned by a call to the *predictor*. For example, in the case of a binary classifier, if the predictor outputs a decision score (i.e. uses the *decision_function* method) which returns one column, then the length of the *target_names* should be one. On the other hand, if the predictor outputs a prediction probability (i.e. uses the *predict_proba* method) which returns two columns (one for the negative class and one for the positive class), then the length of the *target_names* should be two.

```
explain(X, features=None, kind='average', percentiles=(0.0, 1.0), grid_resolution=100, grid_points=None)
```

Calculates the partial dependence for each feature and/or tuples of features with respect to the all targets and the reference dataset *X*.

Parameters

- **X** (ndarray) – A $N \times F$ tabular dataset used to calculate partial dependence curves. This is typically the training dataset or a representative sample.
- **features** (Optional[List[Union[int, Tuple[int, int]]]]) – An optional list of features or tuples of features for which to calculate the partial dependence. If not provided, the partial dependence will be computed for every single features in the dataset. Some example for *features* would be: [0, 2], [0, 2, (0, 2)], [(0, 2)], where 0 and 2 correspond to column 0 and 2 in X, respectively.
- **kind** (Literal['average', 'individual', 'both']) – If set to 'average', then only the partial dependence (PD) averaged across all samples from the dataset is returned. If set to 'individual', then only the individual conditional expectation (ICE) is returned for each data point from the dataset. Otherwise, if set to 'both', then both the PD and the ICE are returned.
- **percentiles** (Tuple[float, float]) – Lower and upper percentiles used to limit the feature values to potentially remove outliers from low-density regions. Note that for features with not many data points with large/low values, the PD estimates are less reliable in those extreme regions. The values must be in [0, 1]. Only used with *grid_resolution*.
- **grid_resolution** (int) – Number of equidistant points to split the range of each target feature. Only applies if the number of unique values of a target feature in the reference dataset X is greater than the *grid_resolution* value. For example, consider a case where a feature can take the following values: [0.1, 0.3, 0.35, 0.351, 0.4, 0.41, 0.44, ..., 0.5, 0.54, 0.56, 0.6, 0.65, 0.7, 0.9], and we are not interested in evaluating the marginal effect at every single point as it can become computationally costly (assume hundreds/thousands of points) without providing any additional information for nearby points (e.g., 0.35 and 351). By setting *grid_resolution*=5, the marginal effect is computed for the values [0.1, 0.3, 0.5, 0.7, 0.9] instead, which is less computationally demanding and can provide similar insights regarding the model's behaviour. Note that the extreme values of the grid can be controlled using the *percentiles* argument.
- **grid_points** (Optional[Dict[int, Union[List, ndarray]]]) – Custom grid points. Must be a *dict* where the keys are the target features indices and the values are monotonically increasing arrays defining the grid points for a numerical feature, and a subset of categorical feature values for a categorical feature. If the *grid_points* are not specified, then the grid will be constructed based on the unique target feature values available in the dataset X, or based on the *grid_resolution* and *percentiles* (check *grid_resolution* to see when it applies). For categorical features, the corresponding value in the *grid_points* can be specified either as array of strings or array of integers corresponding the label encodings. Note that the label encoding must match the ordering of the values provided in the *categorical_names*.

Return type

[Explanation](#)

Returns

explanation – An *Explanation* object containing the data and the metadata of the calculated partial dependence curves. See usage at [Partial dependence examples](#) for details

```
class alibi.explainers.partial_dependence.PartialDependenceBase(predictor, feature_names=None,
                                                                categorical_names=None,
                                                                target_names=None,
                                                                verbose=False)
```

Bases: [Explainer](#), [ABC](#)

__init__(*predictor*, *feature_names*=None, *categorical_names*=None, *target_names*=None, *verbose*=False)

Base class of the partial dependence for tabular datasets. Supports multiple feature interactions.

Parameters

- **predictor** (`Union`[`BaseEstimator`, `Callable`[[`ndarray`], `ndarray`]]) – A *sklearn* estimator or a prediction function which receives as input a *numpy* array of size $N \times F$ and outputs a *numpy* array of size N (i.e. $(N,)$ or $N \times T$, where N is the number of input instances, F is the number of features and T is the number of targets).
- **feature_names** (`Optional`[`List`[`str`]]) – A list of feature names used for displaying results.
- **categorical_names** (`Optional`[`Dict`[`int`, `List`[`str`]]]) – Dictionary where keys are feature columns and values are the categories for the feature. Necessary to identify the categorical features in the dataset. An example for *categorical_names* would be:

```
category_map = {0: ["married", "divorced"], 3: ["high school diploma", "master's degree"]}
```

- **target_names** (`Optional`[`List`[`str`]]) – A list of target/output names used for displaying results.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

explain(*X*, *features*=None, *kind*='average', *percentiles*=(0.0, 1.0), *grid_resolution*=100, *grid_points*=None)

Calculates the partial dependence for each feature and/or tuples of features with respect to the all targets and the reference dataset *X*.

Parameters

- **X** (`ndarray`) – A $N \times F$ tabular dataset used to calculate partial dependence curves. This is typically the training dataset or a representative sample.
- **features** (`Optional`[`List`[`Union`[`int`, `Tuple`[`int`, `int`]]]]) – An optional list of features or tuples of features for which to calculate the partial dependence. If not provided, the partial dependence will be computed for every single features in the dataset. Some example for *features* would be: $[0, 2]$, $[0, 2, (0, 2)]$, $[(0, 2)]$, where 0 and 2 correspond to column 0 and 2 in *X*, respectively.
- **kind** (`Literal`['average', 'individual', 'both']) – If set to 'average', then only the partial dependence (PD) averaged across all samples from the dataset is returned. If set to 'individual', then only the individual conditional expectation (ICE) is returned for each data point from the dataset. Otherwise, if set to 'both', then both the PD and the ICE are returned.
- **percentiles** (`Tuple`[`float`, `float`]) – Lower and upper percentiles used to limit the feature values to potentially remove outliers from low-density regions. Note that for features with not many data points with large/low values, the PD estimates are less reliable in those extreme regions. The values must be in $[0, 1]$. Only used with *grid_resolution*.
- **grid_resolution** (`int`) – Number of equidistant points to split the range of each target feature. Only applies if the number of unique values of a target feature in the reference dataset *X* is greater than the *grid_resolution* value. For example, consider a case where a feature can take the following values: $[0.1, 0.3, 0.35, 0.351, 0.4, 0.41, 0.44, \dots, 0.5, 0.54, 0.56, 0.6, 0.65, 0.7, 0.9]$, and we are not interested in evaluating the marginal effect at every single point as it can become computationally costly (assume hundreds/thousands of points) without providing any additional information for nearby points (e.g., 0.35 and 351). By setting *grid_resolution*=5, the marginal effect is

computed for the values `[0.1, 0.3, 0.5, 0.7, 0.9]` instead, which is less computationally demanding and can provide similar insights regarding the model's behaviour. Note that the extreme values of the grid can be controlled using the *percentiles* argument.

- **grid_points** (`Optional[Dict[int, Union[List, ndarray]]]`) – Custom grid points. Must be a *dict* where the keys are the target features indices and the values are monotonically increasing arrays defining the grid points for a numerical feature, and a subset of categorical feature values for a categorical feature. If the *grid_points* are not specified, then the grid will be constructed based on the unique target feature values available in the dataset *X*, or based on the *grid_resolution* and *percentiles* (check *grid_resolution* to see when it applies). For categorical features, the corresponding value in the *grid_points* can be specified either as array of strings or array of integers corresponding the label encodings. Note that the label encoding must match the ordering of the values provided in the *categorical_names*.

Return type

Explanation

Returns

explanation – An *Explanation* object containing the data and the metadata of the calculated partial dependence curves. See usage at [Partial dependence examples](#) for details

`reset_predictor(predictor)`

Resets the predictor function or tree-based *sklearn* estimator.

Parameters

predictor (`Union[Callable[[ndarray], ndarray], BaseEstimator]`) – New predictor function or tree-based *sklearn* estimator.

Return type

None

```
class alibi.explainers.partial_dependence.TreePartialDependence(predictor, feature_names=None,
                                                                categorical_names=None,
                                                                target_names=None,
                                                                verbose=False)
```

Bases: [PartialDependenceBase](#)

Tree-based model *sklearn* implementation of the partial dependence for tabular datasets. Supports multiple feature interactions. This method is faster than the general black-box implementation but is only supported by some tree-based estimators. The computation is based on a weighted tree traversal. For more details on the computation, check the [sklearn documentation page](#). The supported *sklearn* models are: *GradientBoostingClassifier*, *GradientBoostingRegressor*, *HistGradientBoostingClassifier*, *HistGradientBoostingRegressor*, *HistGradientBoostingRegressor*, *DecisionTreeRegressor*, *RandomForestRegressor*.

__init__ (`predictor, feature_names=None, categorical_names=None, target_names=None, verbose=False`)

Initialize tree-based model *sklearn* implementation of partial dependence.

Parameters

- **predictor** (`BaseEstimator`) – A tree-based *sklearn* estimator.
- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.
- **categorical_names** (`Optional[Dict[int, List[str]]]`) – Dictionary where keys are feature columns and values are the categories for the feature. Necessary to identify the categorical features in the dataset. An example for *categorical_names* would be:

```
category_map = {0: ["married", "divorced"], 3: ["high school diploma", "master's degree"]}
```

- **target_names** (`Optional[List[str]]`) – A list of target/output names used for displaying results.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

Notes

The length of the *target_names* should match the number of columns returned by a call to the *predictor.decision_function*. In the case of a binary classifier, the decision score consists of a single column. Thus, the length of the *target_names* should be one.

explain(*X*, *features=None*, *percentiles=(0.0, 1.0)*, *grid_resolution=100*, *grid_points=None*)

Calculates the partial dependence for each feature and/or tuples of features with respect to the all targets and the reference dataset *X*.

Parameters

- **X** (`ndarray`) – A $N \times F$ tabular dataset used to calculate partial dependence curves. This is typically the training dataset or a representative sample.
- **features** (`Optional[List[Union[int, Tuple[int, int]]]]`) – An optional list of features or tuples of features for which to calculate the partial dependence. If not provided, the partial dependence will be computed for every single features in the dataset. Some example for *features* would be: `[0, 2]`, `[0, 2, (0, 2)]`, `[(0, 2)]`, where 0 and 2 correspond to column 0 and 2 in *X*, respectively.
- **percentiles** (`Tuple[float, float]`) – Lower and upper percentiles used to limit the feature values to potentially remove outliers from low-density regions. Note that for features with not many data points with large/low values, the PD estimates are less reliable in those extreme regions. The values must be in `[0, 1]`. Only used with *grid_resolution*.
- **grid_resolution** (`int`) – Number of equidistant points to split the range of each target feature. Only applies if the number of unique values of a target feature in the reference dataset *X* is greater than the *grid_resolution* value. For example, consider a case where a feature can take the following values: `[0.1, 0.3, 0.35, 0.351, 0.4, 0.41, 0.44, ..., 0.5, 0.54, 0.56, 0.6, 0.65, 0.7, 0.9]`, and we are not interested in evaluating the marginal effect at every single point as it can become computationally costly (assume hundreds/thousands of points) without providing any additional information for nearby points (e.g., 0.35 and 351). By setting *grid_resolution*=5, the marginal effect is computed for the values `[0.1, 0.3, 0.5, 0.7, 0.9]` instead, which is less computationally demanding and can provide similar insights regarding the model's behaviour. Note that the extreme values of the grid can be controlled using the *percentiles* argument.
- **grid_points** (`Optional[Dict[int, Union[List, ndarray]]]`) – Custom grid points. Must be a *dict* where the keys are the target features indices and the values are monotonically increasing arrays defining the grid points for a numerical feature, and a subset of categorical feature values for a categorical feature. If the *grid_points* are not specified, then the grid will be constructed based on the unique target feature values available in the dataset *X*, or based on the *grid_resolution* and *percentiles* (check *grid_resolution* to see when it applies). For categorical features, the corresponding value in the *grid_points* can be specified either as array of strings or array of integers corresponding the label encodings. Note that the label encoding must match the ordering of the values provided in the *categorical_names*.

Return type*Explanation*

```
alibi.explainers.partial_dependence.plot_pd(exp, features='all', target=0, n_cols=3, n_ice=100,
                                           center=False, pd_limits=None, levels=8, ax=None,
                                           sharey='all', pd_num_kw=None, ice_num_kw=None,
                                           pd_cat_kw=None, ice_cat_kw=None,
                                           pd_num_num_kw=None, pd_num_cat_kw=None,
                                           pd_cat_cat_kw=None, fig_kw=None)
```

Plot partial dependence curves on matplotlib axes.

Parameters

- **exp** – An *Explanation* object produced by a call to the `alibi.explainers.partial_dependence.PartialDependence.explain()` method.
- **features** – A list of features entries in the `exp.data['feature_names']` to plot the partial dependence curves for, or 'all' to plot all the explained feature or tuples of features. This includes tuples of features. For example, if `exp.data['feature_names'] = ['temp', 'hum', ('temp', 'windspeed')]` and we want to plot the partial dependence only for the 'temp' and ('temp', 'windspeed'), then we would set `features=[0, 2]`. Defaults to 'all'.
- **target** – The target name or index for which to plot the partial dependence (PD) curves. Can be a mix of integers denoting target index or strings denoting entries in `exp.meta['params']['target_names']`.
- **n_cols** – Number of columns to organize the resulting plot into.
- **n_ice** – Number of ICE plots to be displayed. Can be
 - a string taking the value 'all' to display the ICE curves for every instance in the reference dataset.
 - an integer for which `n_ice` instances from the reference dataset will be sampled uniformly at random to display their ICE curves.
 - a list of integers, where each integer represents an index of an instance in the reference dataset to display their ICE curves.
- **center** – Boolean flag to center the individual conditional expectation (ICE) curves. As mentioned in Goldstein et al. (2014), the heterogeneity in the model can be difficult to discern when the intercepts of the ICE curves cover a wide range. Centering the ICE curves removes the level effects and helps to visualise the heterogeneous effect.
- **pd_limits** – Minimum and maximum y-limits for all the one-way PD plots. If None will be automatically inferred.
- **levels** – Number of levels in the contour plot.
- **ax** – A *matplotlib* axes object or a *numpy* array of *matplotlib* axes to plot on.
- **sharey** – A parameter specifying whether the y-axis of the PD and ICE curves should be on the same scale for several features. Possible values are: 'all' | 'row' | None.
- **pd_num_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the PD for a numerical feature.
- **ice_num_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the ICE for a numerical feature.
- **pd_cat_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the PD for a categorical feature.

- **ice_cat_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the ICE for a categorical feature.
- **pd_num_num_kw** – Keyword arguments passed to the `matplotlib.pyplot.contourf` function when plotting the PD for two numerical features.
- **pd_num_cat_kw** – Keyword arguments passed to the `matplotlib.pyplot.plot` function when plotting the PD for a numerical and a categorical feature.
- **pd_cat_cat_kw** – Keyword arguments passed to the `alibi.utils.visualization.heatmap()` function when plotting the PD for two categorical features.
- **fig_kw** – Keyword arguments passed to the `matplotlib.figure.set` function.

Returns

An array of `plt.Axes` with the resulting partial dependence plots.

alibi.explainers.pd_variance module

```
class alibi.explainers.pd_variance.Method(value)
```

Bases: `str`, `Enum`

Enumeration of supported methods.

```
IMPORTANCE = 'importance'
```

```
INTERACTION = 'interaction'
```

```
class alibi.explainers.pd_variance.PartialDependenceVariance(predictor, feature_names=None,
                                                             categorical_names=None,
                                                             target_names=None,
                                                             verbose=False)
```

Bases: `Explainer`

Implementation of the partial dependence(PD) variance feature importance and feature interaction for tabular datasets. The method measure the importance feature importance as the variance within the PD function. Similar, the potential feature interaction is measured by computing the variance within the two-way PD function by holding one variable constant and letting the other vary. Supports black-box models and the following *sklearn* tree-based models: *GradientBoostingClassifier*, *GradientBoostingRegressor*, *HistGradientBoostingClassifier*, *HistGradientBoostingRegressor*, *DecisionTreeRegressor*, *RandomForestRegressor*.

For details of the method see the original paper: <https://arxiv.org/abs/1805.04755> .

```
__init__(predictor, feature_names=None, categorical_names=None, target_names=None, verbose=False)
```

Initialize black-box/tree-based model implementation for the partial dependence variance feature importance.

Parameters

- **predictor** (`Union[BaseEstimator, Callable[[ndarray], ndarray]]`) – A *sklearn* estimator or a prediction function which receives as input a *numpy* array of size $N \times F$ and outputs a *numpy* array of size N (i.e. $(N,)$ or $N \times T$, where N is the number of input instances, F is the number of features and T is the number of targets).
- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.

- **categorical_names** (`Optional[Dict[int, List[str]]]`) – Dictionary where keys are feature columns and values are the categories for the feature. Necessary to identify the categorical features in the dataset. An example for *categorical_names* would be:

```
category_map = {0: ["married", "divorced"], 3: ["high school diploma", "master's degree"]}
```

- **target_names** (`Optional[List[str]]`) – A list of target/output names used for displaying results.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

Notes

The length of the *target_names* should match the number of columns returned by a call to the *predictor*. For example, in the case of a binary classifier, if the predictor outputs a decision score (i.e. uses the *decision_function* method) which returns one column, then the length of the *target_names* should be one. On the other hand, if the predictor outputs a prediction probability (i.e. uses the *predict_proba* method) which returns two columns (one for the negative class and one for the positive class), then the length of the *target_names* should be two.

explain(*X*, *features=None*, *method='importance'*, *percentiles=(0.0, 1.0)*, *grid_resolution=100*, *grid_points=None*)

Calculates the variance partial dependence feature importance for each feature with respect to the all targets and the reference dataset *X*.

Parameters

- **X** (`ndarray`) – A $N \times F$ tabular dataset used to calculate partial dependence curves. This is typically the training dataset or a representative sample.
- **features** (`Union[List[int], List[Tuple[int, int]], None]`) – A list of features for which to compute the feature importance or a list of feature pairs for which to compute the feature interaction. Some example of *features* would be: `[0, 1, 3]`, `[(0, 1), (0, 3), (1, 3)]`, where 0, 1, and 3 correspond to the columns 0, 1, and 3 in *X*. If not provided, the feature importance or the feature interaction will be computed for every feature or for every combination of feature pairs, depending on the parameter *method*.
- **method** (`Literal['importance', 'interaction']`) – Flag to specify whether to compute the feature importance or the feature interaction of the elements provided in *features*. Supported values: `'importance' | 'interaction'`.
- **percentiles** (`Tuple[float, float]`) – Lower and upper percentiles used to limit the feature values to potentially remove outliers from low-density regions. Note that for features with not many data points with large/low values, the PD estimates are less reliable in those extreme regions. The values must be in `[0, 1]`. Only used with *grid_resolution*.
- **grid_resolution** (`int`) – Number of equidistant points to split the range of each target feature. Only applies if the number of unique values of a target feature in the reference dataset *X* is greater than the *grid_resolution* value. For example, consider a case where a feature can take the following values: `[0.1, 0.3, 0.35, 0.351, 0.4, 0.41, 0.44, ..., 0.5, 0.54, 0.56, 0.6, 0.65, 0.7, 0.9]`, and we are not interested in evaluating the marginal effect at every single point as it can become computationally costly (assume hundreds/thousands of points) without providing any additional information for nearby points (e.g., 0.35 and 351). By setting *grid_resolution*=5, the marginal effect is

computed for the values `[0.1, 0.3, 0.5, 0.7, 0.9]` instead, which is less computationally demanding and can provide similar insights regarding the model's behaviour. Note that the extreme values of the grid can be controlled using the *percentiles* argument.

- **grid_points** (`Optional[Dict[int, Union[List, ndarray]]`) – Custom grid points. Must be a *dict* where the keys are the target features indices and the values are monotonically increasing arrays defining the grid points for a numerical feature, and a subset of categorical feature values for a categorical feature. If the *grid_points* are not specified, then the grid will be constructed based on the unique target feature values available in the dataset *X*, or based on the *grid_resolution* and *percentiles* (check *grid_resolution* to see when it applies). For categorical features, the corresponding value in the *grid_points* can be specified either as array of strings or array of integers corresponding the label encodings. Note that the label encoding must match the ordering of the values provided in the *categorical_names*.

Return type

Explanation

Returns

explanation – An *Explanation* object containing the data and the metadata of the calculated partial dependence curves and feature importance/interaction. See usage at [Partial dependence variance examples](#) for details

```
alibi.explainers.pd_variance.plot_pd_variance(exp, features='all', targets='all', summarise=True,
                                             n_cols=3, sort=True, top_k=None, plot_limits=None,
                                             ax=None, sharey='all', bar_kw=None, line_kw=None,
                                             fig_kw=None)
```

Plot feature importance and feature interaction based on partial dependence curves on *matplotlib* axes.

Parameters

- **exp** (*Explanation*) – An *Explanation* object produced by a call to the [alibi.explainers.pd_variance.PartialDependenceVariance.explain\(\)](#) method.
- **features** (`Union[List[int], Literal['all']]`) – A list of features entries provided in *feature_names* argument to the [alibi.explainers.pd_variance.PartialDependenceVariance.explain\(\)](#) method, or 'all' to plot all the explained features. For example, if *feature_names* = ['temp', 'hum', 'windspeed'] and we want to plot the values only for the 'temp' and 'windspeed', then we would set *features*=`[0, 2]`. Defaults to 'all'.
- **targets** (`Union[List[Union[int, str]], Literal['all']]`) – A target name/index, or a list of target names/indices, for which to plot the feature importance/interaction, or 'all'. Can be a mix of integers denoting target index or strings denoting entries in *exp.meta['params']['target_names']*. By default 'all' to plot the importance for all features or to plot all the feature interactions.
- **summarise** (`bool`) – Whether to plot only the summary of the feature importance/interaction as a bar plot, or plot comprehensive exposition including partial dependence plots and conditional importance plots.
- **n_cols** (`int`) – Number of columns to organize the resulting plot into.
- **sort** (`bool`) – Boolean flag whether to sort the values in descending order.
- **top_k** (`Optional[int]`) – Number of top k values to be displayed if the *sort*=`True`. If not provided, then all values will be displayed.
- **plot_limits** (`Optional[Tuple[float, float]]`) – Minimum and maximum y-limits for all the line plots. If `None` will be automatically inferred.

- **ax** (`Union[Axes, ndarray, None]`) – A *matplotlib* axes object or a *numpy* array of *matplotlib* axes to plot on.
- **sharey** (`Optional[Literal['all', 'row']]`) – A parameter specifying whether the y-axis of the PD and ICE curves should be on the same scale for several features. Possible values are: 'all' | 'row' | None.
- **bar_kw** (`Optional[dict]`) – Keyword arguments passed to the *matplotlib.pyplot.barh* function.
- **line_kw** (`Optional[dict]`) – Keyword arguments passed to the *matplotlib.pyplot.plot* function.
- **fig_kw** (`Optional[dict]`) – Keyword arguments passed to the *matplotlib.figure.set* function.

Returns

plt.Axes with the summary/detailed exposition plot of the feature importance or feature interaction.

alibi.explainers.permutation_importance module

```
class alibi.explainers.permutation_importance.Kind(value)
```

Bases: `str, Enum`

Enumeration of supported kind.

DIFFERENCE = 'difference'

RATIO = 'ratio'

```
alibi.explainers.permutation_importance.LOSS_FNS = {'log_loss': sklearn.metrics.log_loss,
'mean_absolute_error': sklearn.metrics.mean_absolute_error,
'mean_absolute_percentage_error': sklearn.metrics.mean_absolute_percentage_error,
'mean_squared_error': sklearn.metrics.mean_squared_error, 'mean_squared_log_error':
sklearn.metrics.mean_squared_log_error}
```

Dictionary of supported string specified loss functions

- 'mean_absolute_error' - Mean absolute error regression loss. See [sklearn.metrics.mean_absolute_error](#) for documentation.
- 'mean_squared_error' - Mean squared error regression loss. See [sklearn.metrics.mean_squared_error](#) for documentation.
- 'mean_squared_log_error' - Mean squared logarithmic error regression loss. See [sklearn.metrics.mean_squared_log_error](#) for documentation.
- 'mean_absolute_percentage_error' - Mean absolute percentage error (MAPE) regression loss. See [sklearn.metrics.mean_absolute_percentage_error](#) for documentation.
- 'log_loss' - Log loss, aka logistic loss or cross-entropy loss. See [sklearn.metrics.log_loss](#) for documentation.

```
class alibi.explainers.permutation_importance.Method(value)
```

Bases: `str, Enum`

Enumeration of supported method.


```
ESTIMATE = 'estimate'
```

```
EXACT = 'exact'
```

```
class alibi.explainers.permutation_importance.PermutationImportance(predictor, loss_fns=None,
                                                                    score_fns=None,
                                                                    feature_names=None,
                                                                    verbose=False)
```

Bases: [Explainer](#)

Implementation of the permutation feature importance for tabular datasets. The method measure the importance of a feature as the relative increase/decrease in the loss/score function when the feature values are permuted. Supports black-box models.

For details of the method see the papers:

- <https://link.springer.com/article/10.1023/A:1010933404324>
- <https://arxiv.org/abs/1801.01489>

```
__init__(predictor, loss_fns=None, score_fns=None, feature_names=None, verbose=False)
```

Initialize the permutation feature importance.

Parameters

- **predictor** (`Callable[[ndarray], ndarray]`) – A prediction function which receives as input a *numpy* array of size $N \times F$, and outputs a *numpy* array of size N (i.e. $(N,)$ or $N \times T$, where N is the number of input instances, F is the number of features, and T is the number of targets. Note that the output shape must be compatible with the loss and score functions provided in *loss_fns* and *score_fns*.
- **loss_fns** (`Union[Literal['mean_absolute_error', 'mean_squared_error', 'mean_squared_log_error', 'mean_absolute_percentage_error', 'log_loss'], List[Literal['mean_absolute_error', 'mean_squared_error', 'mean_squared_log_error', 'mean_absolute_percentage_error', 'log_loss']], Callable[[ndarray, ndarray, Optional[ndarray]], float], Dict[str, Callable[[ndarray, ndarray, Optional[ndarray]], float]], None)`) – A literal, or a list of literals, or a loss function, or a dictionary of loss functions having as keys the names of the loss functions and as values the loss functions (i.e., lower values are better). The available literal values are described in [alibi.explainers.permutation_importance.LOSS_FNS](#). Note that the *predictor* output must be compatible with every loss function. Every loss function is expected to receive the following arguments:
 - *y_true* : `np.ndarray` - a *numpy* array of ground-truth labels.
 - *y_pred* | *y_score* : `np.ndarray` - a *numpy* array of model predictions. This corresponds to the output of the model.
 - *sample_weight*: `Optional`[`np.ndarray`] - a *numpy* array of sample weights.
- **score_fns** (`Union[Literal['accuracy', 'precision', 'recall', 'f1', 'roc_auc', 'r2'], List[Literal['accuracy', 'precision', 'recall', 'f1', 'roc_auc', 'r2']], Callable[[ndarray, ndarray, Optional[ndarray]], float], Dict[str, Callable[[ndarray, ndarray, Optional[ndarray]], float]], None)`) – A literal, or a list or literals, or a score function, or a dictionary of score functions having as keys the names of the score functions and as values the score functions (i.e, higher values are better). The available literal values are described in [alibi.explainers.permutation_importance.SCORE_FNS](#). As with the *loss_fns*, the *predictor* output

must be compatible with every score function and the score function must have the same signature presented in the *loss_fns* parameter description.

- **feature_names** (`Optional[List[str]]`) – A list of feature names used for displaying results.
- **verbose** (`bool`) – Whether to print the progress of the explainer.

explain(*X*, *y*, *features=None*, *method='estimate'*, *kind='ratio'*, *n_repeats=50*, *sample_weight=None*)

Computes the permutation feature importance for each feature with respect to the given loss or score functions and the dataset (*X*, *y*).

Parameters

- **X** (`ndarray`) – A $N \times F$ input feature dataset used to calculate the permutation feature importance. This is typically the test dataset.
- **y** (`ndarray`) – Ground-truth labels array of size N (i.e. $(N,)$) corresponding the input feature *X*.
- **features** (`Optional[List[Union[int, Tuple[int, ...]]]`) – An optional list of features or tuples of features for which to compute the permutation feature importance. If not provided, the permutation feature importance will be computed for every single features in the dataset. Some example of *features* would be: `[0, 2]`, `[0, 2, (0, 2)]`, `[(0, 2)]`, where 0 and 2 correspond to column 0 and 2 in *X*, respectively.
- **method** (`Literal['estimate', 'exact']`) – The method to be used to compute the feature importance. If set to 'exact', a “switch” operation is performed across all observed pairs, by excluding pairings that are actually observed in the original dataset. This operation is quadratic in the number of samples ($N \times (N - 1)$ samples) and thus can be computationally intensive. If set to 'estimate', the dataset will be divided in half. The values of the first half containing the ground-truth labels the rest of the features (i.e. features that are left intact) is matched with the values of the second half of the permuted features, and the other way around. This method is computationally lighter and provides estimate error bars given by the standard deviation. Note that for some specific loss and score functions, the estimate does not converge to the exact metric value.
- **kind** (`Literal['ratio', 'difference']`) – Whether to report the importance as the loss/score ratio or the loss/score difference. Available values are: 'ratio' | 'difference'.
- **n_repeats** (`int`) – Number of times to permute the feature values. Considered only when *method='estimate'*.
- **sample_weight** (`Optional[ndarray]`) – Optional weight for each sample instance.

Return type

Explanation

Returns

explanation – An *Explanation* object containing the data and the metadata of the permutation feature importance. See usage at [Permutation feature importance examples](#) for details

reset_predictor(*predictor*)

Resets the predictor function.

Parameters

predictor (`Callable`) – New predictor function.

Return type

`None`

```
alibi.explainers.permutation_importance.SCORE_FNS = {'accuracy':
sklearn.metrics.accuracy_score, 'f1': sklearn.metrics.f1_score, 'precision':
sklearn.metrics.precision_score, 'r2': sklearn.metrics.r2_score, 'recall':
sklearn.metrics.recall_score, 'roc_auc': sklearn.metrics.roc_auc_score}
```

Dictionary of supported string specified score functions

- 'accuracy' - Accuracy classification score. See [sklearn.metrics.accuracy_score](#) for documentation.
- 'precision' - Precision score. See [sklearn.metrics.precision_score](#) for documentation.
- 'recall' - Recall score. See [sklearn.metrics.recall_score](#) for documentation.
- 'f1_score' - F1 score. See [sklearn.metrics.f1_score](#) for documentation.
- 'roc_auc_score' - Area Under the Receiver Operating Characteristic Curve (ROC AUC) score. See [sklearn.metrics.roc_auc_score](#) for documentation.
- 'r2_score' - R^2 (coefficient of determination) regression score. See [sklearn.metrics.r2_score](#) for documentation.

```
alibi.explainers.permutation_importance.plot_permutation_importance(exp, features='all',
                                                                    metric_names='all',
                                                                    n_cols=3, sort=True,
                                                                    top_k=None, ax=None,
                                                                    bar_kw=None,
                                                                    fig_kw=None)
```

Plot permutation feature importance on *matplotlib* axes.

Parameters

- **exp** – An *Explanation* object produced by a call to the [alibi.explainers.permutation_importance.PermutationImportance.explain\(\)](#) method.
- **features** – A list of feature entries provided in *feature_names* argument to the [alibi.explainers.permutation_importance.PermutationImportance.explain\(\)](#) method, or 'all' to plot all the explained features. For example, consider that the *feature_names* = ['temp', 'hum', 'windspeed', 'season']. If we set *features=None* in the *explain* method, meaning that all the feature were explained, and we want to plot only the values for the 'temp' and 'windspeed', then we would set *features*=[0, 2]. Otherwise, if we set *features*=[1, 2, 3] in the *explain* method, meaning that we explained ['hum', 'windspeed', 'season'], and we want to plot the values only for ['windspeed', 'season'], then we would set *features*=[1, 2] (i.e., their index in the *features* list passed to the *explain* method). Defaults to 'all'.
- **metric_names** – A list of metric entries in the *exp.data['metrics']* to plot the permutation feature importance for, or 'all' to plot the permutation feature importance for all metrics (i.e., loss and score functions). The ordering is given by the concatenation of the loss metrics followed by the score metrics.
- **n_cols** – Number of columns to organize the resulting plot into.
- **sort** – Boolean flag whether to sort the values in descending order.
- **top_k** – Number of top k values to be displayed if the *sort*=True. If not provided, then all values will be displayed.
- **ax** – A *matplotlib* axes object or a *numpy* array of *matplotlib* axes to plot on.
- **bar_kw** – Keyword arguments passed to the [matplotlib.pyplot.barh](#) function.

- **fig_kw** – Keyword arguments passed to the `matplotlib.figure.set` function.

Returns

`plt.Axes` with the feature importance plot.

alibi.explainers.shap_wrappers module

`alibi.explainers.shap_wrappers.DISTRIBUTED_OPTS: Dict = {'batch_size': 1, 'n_cpus': None}`

Default distributed options for KernelShap:

- `'n_cpus' : int` - number of available CPUs available to parallelize explanations. Performance is significantly boosted when the number specified represents physical CPUs, but small (nonlinear) gains are observed when virtual CPUs are specified. If set to `None`, the code will run sequentially.
- `'batch_size': int`, how many instances are explained in the same remote process at once. The `shap` library of KernelShap is not vectorised, so no significant gains are made by specifying batches. See [blog post](#) for batch size experiments results. If set to `None`, an input array is split in (roughly) equal parts and distributed across the available CPUs.

class `alibi.explainers.shap_wrappers.KernelExplainerWrapper(*args, **kwargs)`

Bases: `KernelExplainer`

A wrapper around `shap.KernelExplainer` that supports:

- fixing the seed when instantiating the `KernelExplainer` in a separate process.
- passing a batch index to the explainer so that a parallel explainer pool can return batches in arbitrary order.

__init__(*args, **kwargs)

Parameters

- ***args** – Arguments and keyword arguments for `shap.KernelExplainer` constructor.
- ****kwargs** – Arguments and keyword arguments for `shap.KernelExplainer` constructor.

get_explanation(X, **kwargs)

Wrapper around `shap.KernelExplainer.shap_values` that allows calling the method with a tuple containing a batch index and a batch of instances.

Parameters

- **X** (`Union[Tuple[int, ndarray], ndarray]`) – When called from a distributed context, it is a tuple containing a batch index and a batch to be explained. Otherwise, it is an array of instances to be explained.
- ****kwargs** – `shap.KernelExplainer.shap_values` kwarg values.

Return type

`Union[Tuple[int, ndarray], Tuple[int, List[ndarray]], ndarray, List[ndarray]]`

return_attribute(name)

Returns an attribute specified by its name. Used in a distributed context where the actor properties cannot be accessed using the dot syntax.

Return type

`Any`

class `alibi.explainers.shap_wrappers.KernelShap(predictor, link='identity', feature_names=None, categorical_names=None, task='classification', seed=None, distributed_opts=None)`

Bases: [Explainer](#), [FitMixin](#)

```
__init__(predictor, link='identity', feature_names=None, categorical_names=None, task='classification',
          seed=None, distributed_opts=None)
```

A wrapper around the `shap.KernelExplainer` class. It extends the current `shap` library functionality by allowing the user to specify variable groups in order to treat one-hot encoded categorical as one during sampling. The user can also specify whether to aggregate the `shap` values estimate for the encoded levels of categorical variables as an optional argument to `explain`, if grouping arguments are not passed to `fit`.

Parameters

- **predictor** ([Callable](#)[[[ndarray](#)], [ndarray](#)]) – A callable that takes as an input a *samples x features* array and outputs a *samples x n_outputs* model outputs. The *n_outputs* should represent model output in margin space. If the model outputs probabilities, then the link should be set to 'logit' to ensure correct force plots.
- **link** ([str](#)) – Valid values are 'identity' or 'logit'. A generalized linear model link to connect the feature importance values to the model output. Since the feature importance values, ϕ , sum up to the model output, it often makes sense to connect them to the output with a link function where $link(output - expected_value) = sum(\phi)$. Therefore, for a model which outputs probabilities, `link='logit'` makes the feature effects have log-odds (evidence) units and `link='identity'` means that the feature effects have probability units. Please see this [example](#) for an in-depth discussion about the semantics of explaining the model in the probability or margin space.
- **feature_names** ([Union](#)[[List](#)[[str](#)], [Tuple](#)[[str](#)], [None](#)]) – Used to infer group names when categorical data is treated by grouping and `group_names` input to `fit` is not specified, assuming it has the same length as the `groups` argument of `fit` method. It is also used to compute the `names` field, which appears as a key in each of the values of `explanation.data['raw']['importances']`.
- **categorical_names** ([Optional](#)[[Dict](#)[[int](#), [List](#)[[str](#)]]]) – Keys are feature column indices in the `background_data` matrix (see `fit`). Each value contains strings with the names of the categories for the feature. Used to select the method for background data summarisation (if specified, subsampling is performed as opposed to k-means clustering). In the future it may be used for visualisation.
- **task** ([str](#)) – Can have values 'classification' and 'regression'. It is only used to set the contents of `explanation.data['raw']['prediction']`
- **seed** ([Optional](#)[[int](#)]) – Fixes the random number stream, which influences which subsets are sampled during shap value estimation.
- **distributed_opts** ([Optional](#)[[Dict](#)]) – A dictionary that controls the algorithm distributed execution. See [alibi.explainers.shap_wrappers.DISTRIBUTED_OPTS](#) documentation for details.

```
explain(X, summarise_result=False, cat_vars_start_idx=None, cat_vars_enc_dim=None, **kwargs)
```

Explains the instances in the array X.

Parameters

- **X** ([Union](#)[[ndarray](#), [DataFrame](#), [spmatrix](#)]) – Instances to be explained.
- **summarise_result** ([bool](#)) – Specifies whether the shap values corresponding to dimensions of encoded categorical variables should be summed so that a single shap value is returned for each categorical variable. Both the start indices of the categorical variables (`cat_vars_start_idx`) and the encoding dimensions (`cat_vars_enc_dim`) have to be specified

- **cat_vars_start_idx** (`Optional[Sequence[int]]`) – The start indices of the categorical variables. If specified, `cat_vars_enc_dim` should also be specified.
- **cat_vars_enc_dim** (`Optional[Sequence[int]]`) – The length of the encoding dimension for each categorical variable. If specified `cat_vars_start_idx` should also be specified.
- ****kwargs** – Keyword arguments specifying explain behaviour. Valid arguments are:
 - `nsamples` - controls the number of predictor calls and therefore runtime.
 - `ll_reg` - the algorithm is exponential in the feature dimension. If set to `auto` the algorithm will first run a feature selection algorithm to select the top features, provided the fraction of sampled sets of missing features is less than 0.2 from the number of total subsets. The Akaike Information Criterion is used in this case. See our examples for more details about available settings for this parameter. Note that by first running a feature selection step, the shapley values of the remainder of the features will be different to those estimated from the entire set.

For more details, please see the shap library [documentation](#) .

Return type

Explanation

Returns

explanation – An explanation object containing the shap values and prediction in the `data` field, along with a `meta` field containing additional data. See usage at [KernelSHAP examples](#) for details.

fit(`background_data`, `summarise_background=False`, `n_background_samples=300`, `group_names=None`, `groups=None`, `weights=None`, `**kwargs`)

This takes a background dataset (usually a subsample of the training set) as an input along with several user specified options and initialises a *KernelShap* explainer. The runtime of the algorithm depends on the number of samples in this dataset and on the number of features in the dataset. To reduce the size of the dataset, the `summarise_background` option and `n_background_samples` should be used. To reduce the feature dimensionality, encoded categorical variables can be treated as one during the feature perturbation process; this decreases the effective feature dimensionality, can reduce the variance of the shap values estimation and reduces slightly the number of calls to the predictor. Further runtime savings can be achieved by changing the `nsamples` parameter in the call to `explain`. Runtime reduction comes with an accuracy trade-off, so it is better to experiment with a runtime reduction method and understand results stability before using the system.

Parameters

- **background_data** (`Union[ndarray, spmatrix, DataFrame, Data]`) – Data used to estimate feature contributions and baseline values for force plots. The rows of the background data should represent samples and the columns features.
- **summarise_background** (`Union[bool, str]`) – A large background dataset impacts the runtime and memory footprint of the algorithm. By setting this argument to `True`, only `n_background_samples` from the provided data are selected. If `group_names` or `groups` arguments are specified, the algorithm assumes that the data contains categorical variables so the records are selected uniformly at random. Otherwise, `shap.kmeans` (a wrapper around `sklearn` k-means implementation) is used for selection. If set to `'auto'`, a default of `KERNEL_SHAP_BACKGROUND_THRESHOLD` samples is selected.
- **n_background_samples** (`int`) – The number of samples to keep in the background dataset if `summarise_background=True`.

- **groups** (`Optional[List[Union[Tuple[int], List[int]]]]`) – A list containing sub-lists specifying the indices of features belonging to the same group.
- **group_names** (`Union[List[str], Tuple[str], None]`) – If specified, this array is used to treat groups of features as one during feature perturbation. This feature can be useful, for example, to treat encoded categorical variables as one and can result in computational savings (this may require adjusting the *nsamples* parameter).
- **weights** (`Union[List[float], Tuple[float], ndarray, None]`) – A sequence or array of weights. This is used only if grouping is specified and assigns a weight to each point in the dataset.
- ****kwargs** – Expected keyword arguments include *keep_index* (bool) and should be used if a data frame containing an index column is passed to the algorithm.

Return type*KernelShap***reset_predictor**(*predictor*)

Resets the prediction function.

Parameters**predictor** (*Callable*) – New prediction function.**Return type***None*

```
class alibi.explainers.shap_wrappers.TreeShap(predictor, model_output='raw', feature_names=None,
                                              categorical_names=None, task='classification',
                                              seed=None)
```

Bases: *Explainer*, *FitMixin*

```
__init__(predictor, model_output='raw', feature_names=None, categorical_names=None,
         task='classification', seed=None)
```

A wrapper around the *shap.TreeExplainer* class. It adds the following functionality:

1. Input summarisation options to allow control over background dataset size and hence runtime
2. Output summarisation for sklearn models with one-hot encoded categorical variables.

Users are strongly encouraged to familiarise themselves with the algorithm by reading the method overview in the documentation.

Parameters

- **predictor** (*Any*) – A fitted model to be explained. *XGBoost*, *LightGBM*, *CatBoost* and most tree-based *scikit-learn* models are supported. In the future, *Pyspark* could also be supported. Please open an issue if this is a use case for you.
- **model_output** (*str*) – Supported values are: 'raw', 'probability', 'probability_doubled', 'log_loss':
 - 'raw' - the raw model of the output, which varies by task, is explained. This option should always be used if the *fit* is called without arguments. It should also be set to compute shap interaction values. For regression models it is the standard output, for binary classification in *XGBoost* it is the log odds ratio.
 - 'probability' - the probability output is explained. This option should only be used if *fit* was called with the *background_data* argument set. The effect of specifying this parameter is that the *shap* library will use this information to transform the shap values computed in margin space (aka using the raw output) to shap values that sum to the probability output by the model plus the model expected output

probability. This requires knowledge of the type of output for *predictor* which is inferred by the *shap* library from the model type (e.g., most sklearn models with exception of *sklearn.tree.DecisionTreeClassifier*, *sklearn.ensemble.RandomForestClassifier*, *sklearn.ensemble.ExtraTreesClassifier* output logits) or on the basis of the mapping implemented in the *shap.TreeEnsemble* constructor. Only trees that output log odds and probabilities are supported currently.

- 'probability_doubled' - used for binary classification problem in situations where the model outputs the logits/probabilities for the positive class but shap values for both outcomes are desired. This option should be used only if *fit* was called with the *background_data* argument set. In this case the expected value for the negative class is 1 - expected_value for positive class and the shap values for the negative class are the negative values of the positive class shap values. As before, the explanation happens in the margin space, and the shap values are subsequently adjusted. convert the model output to probabilities. The same considerations as for *probability* apply for this output type too.
- 'log_loss' - logarithmic loss is explained. This option should be used only if *fit* was called with the *background_data* argument set and requires specifying labels, *y*, when calling *explain*. If the objective is squared error, then the transformation $(output - y)^2$ is applied. For binary cross-entropy objective, the transformation $\log(1 + \exp(output)) - y * output$ with $y \in \{0, 1\}$. Currently only binary cross-entropy and squared error losses can be explained.
- **feature_names** (`Union[List[str], Tuple[str], None]`) – Used to compute the *names* field, which appears as a key in each of the values of the *importances* sub-field of the response *raw* field.
- **categorical_names** (`Optional[Dict[int, List[str]]]`) – Keys are feature column indices. Each value contains strings with the names of the categories for the feature. Used to select the method for background data summarisation (if specified, subsampling is performed as opposed to kmeans clustering). In the future it may be used for visualisation.
- **task** (`str`) – Can have values 'classification' and 'regression'. It is only used to set the contents of the *prediction* field in the *data['raw']* response field.

Notes

Tree SHAP is an additive attribution method so it is best suited to explaining output in margin space (the entire real line). For discussion related to explaining models in output vs probability space, please consult this [resource](#).

explain(*X*, *y=None*, *interactions=False*, *approximate=False*, *check_additivity=True*, *tree_limit=None*, *summarise_result=False*, *cat_vars_start_idx=None*, *cat_vars_enc_dim=None*, ***kwargs*)

Explains the instances in *X*. *y* should be passed if the model loss function is to be explained, which can be useful in order to understand how various features affect model performance over time. This is only possible if the explainer has been fitted with a background dataset and requires setting *model_output='log_loss'*.

Parameters

- **X** (`Union[ndarray, DataFrame, Pool]`) – Instances to be explained.
- **y** (`Optional[ndarray]`) – Labels corresponding to rows of *X*. Should be passed only if a background dataset was passed to the *fit* method.
- **interactions** (`bool`) – If *True*, the shap value for every feature of every instance in *X* is decomposed into *X.shape[1] - 1* shap value interactions and one main effect. This is only supported if *fit* is called with *background_dataset=None*.

- **approximate** (`bool`) – If `True`, an approximation to the shap values that does not account for feature order is computed. This was proposed by [Ando Sabaas](#) here . Check [this](#) resource for more details. This option is currently only supported for *xgboost* and *sklearn* models.
- **check_additivity** (`bool`) – If `True`, output correctness is ensured if `model_output='raw'` has been passed to the constructor.
- **tree_limit** (`Optional[int]`) – Explain the output of a subset of the first *tree_limit* trees in an ensemble model.
- **summarise_result** (`bool`) – This should be set to `True` only when some of the columns in *X* represent encoded dimensions of a categorical variable and one single shap value per categorical variable is desired. Both *cat_vars_start_idx* and *cat_vars_enc_dim* should be specified as detailed below to allow this.
- **cat_vars_start_idx** (`Optional[Sequence[int]]`) – The start indices of the categorical variables.
- **cat_vars_enc_dim** (`Optional[Sequence[int]]`) – The length of the encoding dimension for each categorical variable.

Return type*Explanation***Returns**

explanation – An *Explanation* object containing the shap values and prediction in the *data* field, along with a *meta* field containing additional data. See usage at [TreeSHAP examples](#) for details.

fit(*background_data=None, summarise_background=False, n_background_samples=1000, **kwargs*)

This function instantiates an explainer which can then be use to explain instances using the *explain* method. If no background dataset is passed, the explainer uses the path-dependent feature perturbation algorithm to explain the values. As such, only the model raw output can be explained and this should be reflected by passing `model_output='raw'` when instantiating the explainer. If a background dataset is passed, the interventional feature perturbation algorithm is used. Using this algorithm, probability outputs can also be explained. Additionally, if the `model_output='log_loss'` option is passed to the explainer constructor, then the model loss function can be explained by passing the labels as the *y* argument to the *explain* method. A limited number of loss functions are supported, as detailed in the constructor documentation.

Parameters

- **background_data** (`Union[ndarray, DataFrame, None]`) – Data used to estimate feature contributions and baseline values for force plots. The rows of the background data should represent samples and the columns features.
- **summarise_background** (`Union[bool, str]`) – A large background dataset may impact the runtime and memory footprint of the algorithm. By setting this argument to `True`, only *n_background_samples* from the provided data are selected. If the *categorical_names* argument has been passed to the constructor, subsampling of the data is used. Otherwise, *shap.kmeans* (a wrapper around *sklearn.kmeans* implementation) is used for selection. If set to `'auto'`, a default of *TREE_SHAP_BACKGROUND_WARNING_THRESHOLD* samples is selected.
- **n_background_samples** (`int`) – The number of samples to keep in the background dataset if `summarise_background=True`.

Return type*TreeShap*

reset_predictor(*predictor*)

Resets the predictor.

Parameters

predictor (*Any*) – New prediction.

Return type

None

`alibi.explainers.shap_wrappers.rank_by_importance(shap_values, feature_names=None)`

Given the shap values estimated for a multi-output model, this function ranks features according to their importance. The feature importance is the average absolute value for a given feature.

Parameters

- **shap_values** (*List*[*ndarray*]) – Each element corresponds to a *samples x features* array of shap values corresponding to each model output.
- **feature_names** (*Union*[*List*[*str*], *Tuple*[*str*], *None*]) – Each element is the name of the column with the corresponding index in each of the arrays in the *shap_values* list.

Return type

Dict

Returns

importances –

A dictionary of the form:

```
{
    '0': {'ranked_effect': array([0.2, 0.5, ...]), 'names': ['feat_3',
↪ 'feat_5', ...]},
    '1': {'ranked_effect': array([0.3, 0.2, ...]), 'names': ['feat_6',
↪ 'feat_1', ...]},
    ...
    'aggregated': {'ranked_effect': array([0.9, 0.7, ...]), 'names': [
↪ 'feat_3', 'feat_6', ...]}
}
```

The keys of the first level represent the index of the model output. The feature effects in *ranked_effect* and the corresponding feature names in *names* are sorted from highest (most important) to lowest (least important). The values in the *aggregated* field are obtained by summing the shap values for all the model outputs and then computing the effects. Given an output, the effects are defined as the average magnitude of the shap values across the instances to be explained.

`alibi.explainers.shap_wrappers.sum_categories(values, start_idx, enc_feat_dim)`

This function is used to reduce specified slices in a two- or three- dimensional array.

For two-dimensional *values* arrays, for each entry in *start_idx*, the function sums the following *k* columns where *k* is the corresponding entry in the *enc_feat_dim* sequence. The columns whose indices are not in *start_idx* are left unchanged. This arises when the slices contain the shap values for each dimension of an encoded categorical variable and a single shap value for each variable is desired.

For three-dimensional *values* arrays, the reduction is applied for each rank 2 subarray, first along the column dimension and then across the row dimension. This arises when summarising shap interaction values. Each rank 2 array is a *E x E* matrix of shap interaction values, where *E* is the dimension of the data after one-hot encoding. The result of applying the reduction yields a rank 2 array of dimension *F x F*, where *F* is the number of features (i.e., the feature dimension of the data matrix before encoding). By applying this transformation, a single value

describing the interaction of categorical features i and j and a single value describing the interaction of j and i is returned.

Parameters

- **values** (ndarray) – A two or three dimensional array to be reduced, as described above.
- **start_idx** (Sequence[int]) – The start indices of the columns to be summed.
- **enc_feat_dim** (Sequence[int]) – The number of columns to be summed, one for each start index.

Returns

new_values – An array whose columns have been summed according to the entries in *start_idx* and *enc_feat_dim*.

alibi.models package**Subpackages****alibi.models.pytorch package****Submodules****alibi.models.pytorch.actor_critic module**

This module contains the Pytorch implementation of actor-critic networks used in the Counterfactual with Reinforcement Learning for both data modalities. The models' architectures follow the standard actor-critic design and can have broader use-cases.

class alibi.models.pytorch.actor_critic.**Actor**(*hidden_dim*, *output_dim*)

Bases: Module

Actor network. The network follows the standard actor-critic architecture used in Deep Reinforcement Learning. The model is used in Counterfactual with Reinforcement Learning (CFRL) for both data modalities (images and tabular). The hidden dimension used for the all experiments is 256, which is a common choice in most benchmarks.

__init__(*hidden_dim*, *output_dim*)

Constructor.

Parameters

- **hidden_dim** (int) – Hidden dimension.
- **output_dim** (int) – Output dimension

forward(*x*)

Forward pass

Parameters

x (Tensor) – Input tensor.

Return type

Tensor

Returns

Continuous action.

```
class alibi.models.pytorch.actor_critic.Critic(hidden_dim)
```

Bases: `Module`

Critic network. The network follows the standard actor-critic architecture used in Deep Reinforcement Learning. The model is used in Counterfactual with Reinforcement Learning (CFRL) for both data modalities (images and tabular). The hidden dimension used for the all experiments is 256, which is a common choice in most benchmarks.

```
__init__(hidden_dim)
```

Constructor.

Parameters

hidden_dim (`int`) – Hidden dimension.

```
forward(x)
```

Forward pass.

Parameters

x (`Tensor`) – Input tensor.

Return type

`Tensor`

Returns

Critic value.

alibi.models.pytorch.autoencoder module

This module contains a Pytorch general implementation of an autoencoder, by combining the encoder and the decoder module. In addition it provides an implementation of a heterogeneous autoencoder which includes a type checking of the output.

```
class alibi.models.pytorch.autoencoder.AE(encoder, decoder, **kwargs)
```

Bases: `Model`

Autoencoder. Standard autoencoder architecture. The model is composed from two submodules, the encoder and the decoder. The forward pass consist of passing the input to the encoder, obtain the input embedding and pass the embedding through the decoder. The abstraction can be used for multiple data modalities.

```
__init__(encoder, decoder, **kwargs)
```

Constructor. Combine encoder and decoder in AE.

Parameters

- **encoder** (`Module`) – Encoder network.
- **decoder** (`Module`) – Decoder network.

```
forward(x)
```

Forward pass.

Parameters

x (`Tensor`) – Input tensor.

Return type

`Union[Tensor, List[Tensor]]`

Returns

x_hat – Reconstruction of the input tensor.

class `alibi.models.pytorch.autoencoder.HeAE`(*encoder, decoder, **kwargs*)

Bases: [AE](#)

Heterogeneous autoencoder. The model follows the standard autoencoder architecture and includes an additional type check to ensure that the output of the model is a list of tensors. For more details, see [alibi.models.pytorch.autoencoder.AE](#).

__init__(*encoder, decoder, **kwargs*)

Constructor. Combine encoder and decoder in HeAE.

Parameters

- **encoder** (Module) – Encoder network.
- **decoder** (Module) – Decoder network.

forward(*x*)

Forward pass.

Parameters

x (Tensor) – Input tensor.

Return type

[List](#)[Tensor]

Returns

List of reconstruction of the input tensor. First element corresponds to the reconstruction of all the numerical features if they exist, and the rest of the elements correspond to each categorical feature.

alibi.models.pytorch.cfml_models module

This module contains the Pytorch implementation of models used for the Counterfactual with Reinforcement Learning experiments for both data modalities (image and tabular).

class `alibi.models.pytorch.cfml_models.ADULTDecoder`(*hidden_dim, output_dims*)

Bases: Module

ADULT decoder used in the Counterfactual with Reinforcement Learning experiments. The model consists of a fully connected layer with ReLU nonlinearity, and a multiheaded layer, one for each categorical feature and a single head for the rest of numerical features. The hidden dimension used in the paper is 128.

__init__(*hidden_dim, output_dims*)

Constructor.

Parameters

- **hidden_dim** ([int](#)) – Hidden dimension.
- **output_dims** ([List](#)[[int](#)]) – List of output dimensions.

forward(*x*)

Forward pass.

Parameters

x (Tensor) – Input tensor.

Return type

[List](#)[Tensor]

Returns

List of reconstruction of the input tensor. First element corresponds to the reconstruction of all the numerical features if they exist, and the rest of the elements correspond to each categorical feature.

class `alibi.models.pytorch.cfml_models.ADULTEncoder(hidden_dim, latent_dim)`

Bases: `Module`

ADULT encoder used in the Counterfactual with Reinforcement Learning experiments. The model consists of two fully connected layers with ReLU and tanh nonlinearities. The tanh nonlinearity clips the embedding in $[-1, 1]$ as required in the DDPG algorithm (e.g., `[act_low, act_high]`). The layers' dimensions used in the paper are 128 and 15, although those can vary as they were selected to generalize across many datasets.

__init__(*hidden_dim, latent_dim*)

Constructor.

Parameters

- **hidden_dim** (`int`) – Hidden dimension.
- **latent_dim** (`int`) – Latent dimension.

forward(*x*)

Forward pass.

Parameters

x (`Tensor`) – Input tensor.

Return type

`Tensor`

Returns

Encoding representation having each component in the interval $[-1, 1]$

class `alibi.models.pytorch.cfml_models.MNISTClassifier(output_dim)`

Bases: `Model`

MNIST classifier used in the experiments for Counterfactual with Reinforcement Learning. The model consists of two convolutional layers having 64 and 32 channels and a kernel size of 2 with ReLU nonlinearities, followed by maxpooling of size 2 and dropout of 0.3. The convolutional block is followed by a fully connected layer of 256 with ReLU nonlinearity, and finally a fully connected layer is used to predict the class logits (10 in MNIST case).

__init__(*output_dim*)

Constructor.

Parameters

output_dim (`int`) – Output dimension.

forward(*x*)

Forward pass.

Parameters

x (`Tensor`) – Input tensor.

Return type

`Tensor`

Returns

Classification logits.

```
class alibi.models.pytorch.cfrl_models.MNISTDecoder(latent_dim)
```

Bases: Module

MNIST decoder used in the Counterfactual with Reinforcement Learning experiments. The model consists of a fully connected layer of 128 units with ReLU activation followed by a convolutional block. The convolutional block consists of 4 convolutional layers having 8, 8, 8 and 1 channels and a kernel size of 3. Each convolutional layer, except the last one, has ReLU nonlinearities and is followed by an upsampling layer of size 2. The final layers uses a sigmoid activation to clip the output values in $[0, 1]$.

```
__init__(latent_dim)
```

Constructor.

Parameters

latent_dim (`int`) – Latent dimension.

```
forward(x)
```

Forward pass.

Parameters

x (Tensor) – Input tensor.

Return type

Tensor

Returns

Decoded input having each component in the interval $[0, 1]$.

```
class alibi.models.pytorch.cfrl_models.MNISTEncoder(latent_dim)
```

Bases: Module

MNIST encoder used in the experiments for the Counterfactual with Reinforcement Learning. The model consists of 3 convolutional layers having 16, 8 and 8 channels and a kernel size of 3, with ReLU nonlinearities. Each convolutional layer is followed by a maxpooling layer of size 2. Finally, a fully connected layer follows the convolutional block with a tanh nonlinearity. The tanh clips the output between $[-1, 1]$, required in the DDPG algorithm (e.g., `[act_low, act_high]`). The embedding dimension used in the paper is 32, although this can vary.

```
__init__(latent_dim)
```

Constructor.

Parameters

latent_dim (`int`) – Latent dimension.

```
forward(x)
```

Forward pass.

Parameters

x (Tensor) – Input tensor.

Return type

Tensor

Returns

Encoding representation having each component in the interval $[-1, 1]$

alibi.models.pytorch.metrics module

This module contains a loss wrapper and a definition of various monitoring metrics used during training. The model to be trained inherits from `alibi.explainers.models.pytorch.model.Model` and represents a simplified version of the `tensorflow.keras` API for training and monitoring the model. Currently it is used internally to test the functionalities for the Pytorch backend. To be discussed if the module will be exposed to the user in future versions.

class `alibi.models.pytorch.metrics.AccuracyMetric(name='accuracy')`

Bases: `Metric`

Accuracy monitoring metric.

compute_metric(`y_pred`, `y_true`)

Computes accuracy metric given the predicted label and the true label.

Parameters

- **y_pred** (`Union[Tensor, ndarray]`) – Predicted label.
- **y_true** (`Union[Tensor, ndarray]`) – True label.

Return type

`None`

class `alibi.models.pytorch.metrics.LossContainer(loss, name)`

Bases: `object`

Loss wrapped to monitor the average loss throughout training.

__call__(`y_pred`, `y_true`)

Computes and accumulates the loss given the prediction labels and the true labels.

Parameters

- **y_pred** (`Tensor`) – Prediction labels.
- **y_true** (`Tensor`) – True labels.

Return type

`Tensor`

Returns

Loss value.

__init__(`loss`, `name`)

Constructor.

Parameters

- **loss** (`Callable[[Tensor, Tensor], Tensor]`) – Loss function.
- **name** (`str`) – Name of the loss function

reset()

Resets the loss.

result()

Computes the average loss obtain by dividing the cumulated loss by the number of steps

Return type

`Dict[str, float]`

Returns

Average loss.

```
class alibi.models.pytorch.metrics.Metric(reduction=Reduction.MEAN, name='unknown')
```

Bases: [ABC](#)

Monitoring metric object. Supports two types of reduction: mean and sum.

```
__init__(reduction=Reduction.MEAN, name='unknown')
```

Constructor.

Parameters

- **reduction** ([Reduction](#)) – Metric’s reduction type. Possible values *mean* | *sum*. By default *mean*.
- **name** ([str](#)) – Name of the metric.

```
abstract compute_metric(y_pred, y_true)
```

```
reset()
```

Resets the monitoring metric.

```
result()
```

Computes the result according to the reduction procedure.

Return type

[Dict](#)[[str](#), [float](#)]

Returns

Monitoring metric.

```
update_state(values)
```

Update the state of the metric by summing up the metric values and updating the counts by adding the number of instances for which the metric was computed (first dimension).

```
class alibi.models.pytorch.metrics.Reduction(value)
```

Bases: [Enum](#)

Reduction operation supported by the monitoring metrics.

```
MEAN: str = 'mean'
```

```
SUM: str = 'sum'
```

alibi.models.pytorch.model module

This module tries to provided a class wrapper to mimic the TensorFlow API of *tensorflow.keras.Model*. It is intended to simplify the training of a model through methods like `compile`, `fit` and `evaluate` which allow the user to define custom loss functions, optimizers, evaluation metrics, train a model and evaluate it. Currently it is used internally to test the functionalities for the Pytorch backend. To be discussed if the module will be exposed to the user in future versions.

```
class alibi.models.pytorch.model.Model(*args: Any, **kwargs: Any)
```

Bases: [Module](#)

```
compile(optimizer, loss, loss_weights=None, metrics=None)
```

Compiles a model by setting the optimizer and the loss functions, loss weights and metrics to monitor the training of the model.

Parameters

- **optimizer** ([Optimizer](#)) – Optimizer to be used.

- **loss** (`Union[Callable, List[Callable]]`) – Loss function to be used. Can be a list of the loss function which will be weighted and summed up to compute the total loss.
- **loss_weights** (`Optional[List[float]]`) – Weights corresponding to each loss function. Only used if the *loss* argument is a list.
- **metrics** (`Optional[List[Metric]]`) – Metrics used to monitor the training process.

compute_loss(*y_pred*, *y_true*)

Computes the loss given the prediction labels and the true labels.

Parameters

- **y_pred** (`Union[Tensor, List[Tensor]]`) – Prediction labels.
- **y_true** (`Union[Tensor, List[Tensor]]`) – True labels.

Return type

`Tuple[Tensor, Dict[str, float]]`

Returns

A tuple consisting of the total loss computed as a weighted sum of individual losses and a dictionary of individual losses used of logging.

compute_metrics(*y_pred*, *y_true*)

Computes the metrics given the prediction labels and the true labels.

Parameters

- **y_pred** (`Union[Tensor, List[Tensor]]`) – Prediction labels.
- **y_true** (`Union[Tensor, List[Tensor]]`) – True labels.

Return type

`Dict[str, float]`

evaluate(*testloader*)

Evaluation function. The function reports the evaluation metrics used for monitoring the training loop.

Parameters

testloader (`DataLoader`) – Test dataloader.

Return type

`Dict[str, float]`

Returns

Evaluation metrics.

fit(*trainloader*, *epochs*)

Fit method. Equivalent of a training loop.

Parameters

- **trainloader** (`DataLoader`) – Training data loader.
- **epochs** (`int`) – Number of epochs to train the model.

Return type

`Dict[str, float]`

Returns

Final epoch monitoring metrics.

load_weights(*path*)

Loads the weight of the current model.

Return type

`None`

save_weights(*path*)

Save the weight of the current model.

Return type

`None`

test_step(*x*, *y*)

Performs a test step.

Parameters

- **x** (`Tensor`) – Input tensor.
- **y** (`Union[Tensor, List[Tensor]]`) – Label tensor.

train_step(*x*, *y*)

Performs a train step.

Parameters

- **x** (`Tensor`) – Input tensor.
- **y** (`Union[Tensor, List[Tensor]]`) – Label tensor.

Return type

`Dict[str, float]`

validate_prediction_labels(*y_pred*, *y_true*)

Validates the loss functions, loss weights, training labels and prediction labels.

Parameters

- **y_pred** (`Union[Tensor, List[Tensor]]`) – Prediction labels.
- **y_true** (`Union[Tensor, List[Tensor]]`) – True labels.

alibi.models.tensorflow package

Submodules

alibi.models.tensorflow.actor_critic module

This module contains the Tensorflow implementation of actor-critic networks used in the Counterfactual with Reinforcement Learning for both data modalities. The models' architectures follow the standard actor-critic design and can have broader use-cases.

class `alibi.models.tensorflow.actor_critic.Actor`(*hidden_dim*, *output_dim*, ***kwargs*)

Bases: `Model`

Actor network. The network follows the standard actor-critic architecture used in Deep Reinforcement Learning. The model is used in Counterfactual with Reinforcement Learning (CFRL) for both data modalities (images and tabular). The hidden dimension used for the all experiments is 256, which is a common choice in most benchmarks.

```
__init__(hidden_dim, output_dim, **kwargs)
```

Constructor.

Parameters

- **hidden_dim** (`int`) – Hidden dimension
- **output_dim** (`int`) – Output dimension

```
call(x, **kwargs)
```

Forward pass.

Parameters

- **x** (`Tensor`) – Input tensor.
- ****kwargs** – Other arguments. Not used.

Return type

`Tensor`

Returns

Continuous action.

```
class alibi.models.tensorflow.actor_critic.Critic(hidden_dim, **kwargs)
```

Bases: `Model`

Critic network. The network follows the standard actor-critic architecture used in Deep Reinforcement Learning. The model is used in Counterfactual with Reinforcement Learning (CFRL) for both data modalities (images and tabular). The hidden dimension used for the all experiments is 256, which is a common choice in most benchmarks.

```
__init__(hidden_dim, **kwargs)
```

Constructor.

Parameters

- **hidden_dim** (`int`) – Hidden dimension.

```
call(x, **kwargs)
```

Forward pass.

Parameters

- **x** (`Tensor`) – Input tensor.

Return type

`Tensor`

Returns

Critic value.

alibi.models.tensorflow.autoencoder module

This module contains a Tensorflow general implementation of an autoencoder, by combining the encoder and the decoder module. In addition it provides an implementation of a heterogeneous autoencoder which includes a type checking of the output.

```
class alibi.models.tensorflow.autoencoder.AE(encoder, decoder, **kwargs)
```

Bases: `Model`

Autoencoder. Standard autoencoder architecture. The model is composed from two submodules, the encoder and the decoder. The forward pass consists of passing the input to the encoder, obtain the input embedding and pass the embedding through the decoder. The abstraction can be used for multiple data modalities.

__init__(*encoder, decoder, **kwargs*)

Constructor. Combine encoder and decoder in AE

Parameters

- **encoder** (Model) – Encoder network.
- **decoder** (Model) – Decoder network.

call(*x, **kwargs*)

Forward pass.

Parameters

- **x** (Tensor) – Input tensor.
- ****kwargs** – Other arguments passed to encoder/decoder *call* method.

Return type

`Union[Tensor, List[Tensor]]`

Returns

x_hat – Reconstruction of the input tensor.

class `alibi.models.tensorflow.autoencoder.HeAE`(*encoder, decoder, **kwargs*)

Bases: [AE](#)

Heterogeneous autoencoder. The model follows the standard autoencoder architecture and includes an additional type check to ensure that the output of the model is a list of tensors. For more details, see [alibi.models.pytorch.autoencoder.AE](#).

__init__(*encoder, decoder, **kwargs*)

Constructor. Combine encoder and decoder in HeAE.

Parameters

- **encoder** (Model) – Encoder network.
- **decoder** (Model) – Decoder network.

build(*input_shape*)

Build method.

Parameters

input_shape (`Tuple[int, ...]`) – Tensor's input shape.

Return type

`None`

call(*x, **kwargs*)

Forward pass.

Parameters

- **x** (Tensor) – Input tensor.
- ****kwargs** – Other arguments passed to the encoder/decoder.

Return type

`List[Tensor]`

Returns

List of reconstruction of the input tensor. First element corresponds to the reconstruction of all the numerical features if they exist, and the rest of the elements correspond to each categorical feature.

alibi.models.tensorflow.cfml_models module

This module contains the Tensorflow implementation of models used for the Counterfactual with Reinforcement Learning experiments for both data modalities (image and tabular).

class alibi.models.tensorflow.cfml_models.**ADULTDecoder**(hidden_dim, output_dims, **kwargs)

Bases: Model

ADULT decoder used in the Counterfactual with Reinforcement Learning experiments. The model consists of a fully connected layer with ReLU nonlinearity, and a multiheaded layer, one for each categorical feature and a single head for the rest of numerical features. The hidden dimension used in the paper is 128.

__init__(hidden_dim, output_dims, **kwargs)

Constructor.

Parameters

- **hidden_dim** (int) – Hidden dimension.
- **output_dim** – List of output dimensions.

call(x, **kwargs)

Forward pass.

Parameters

- **x** (Tensor) – Input tensor.
- ****kwargs** – Other arguments. Not used.

Return type

List[Tensor]

Returns

List of reconstruction of the input tensor. First element corresponds to the reconstruction of all the numerical features if they exist, and the rest of the elements correspond to each categorical feature.

class alibi.models.tensorflow.cfml_models.**ADULTEncoder**(hidden_dim, latent_dim, **kwargs)

Bases: Model

ADULT encoder used in the Counterfactual with Reinforcement Learning experiments. The model consists of two fully connected layers with ReLU and tanh nonlinearities. The tanh nonlinearity clips the embedding in [-1, 1] as required in the DDPG algorithm (e.g., [act_low, act_high]). The layers' dimensions used in the paper are 128 and 15, although those can vary as they were selected to generalize across many datasets.

__init__(hidden_dim, latent_dim, **kwargs)

Constructor.

Parameters

- **hidden_dim** (int) – Hidden dimension.
- **latent_dim** (int) – Latent dimension.

call(*x*, ****kwargs**)

Forward pass.

Parameters

- **x** (Tensor) – Input tensor.
- ****kwargs** – Other arguments.

Return type

Tensor

Returns

Encoding representation having each component in the interval [-1, 1].

class alibi.models.tensorflow.cfml_models.**MNISTClassifier**(*output_dim=10*, ****kwargs**)

Bases: Model

MNIST classifier used in the experiments for Counterfactual with Reinforcement Learning. The model consists of two convolutional layers having 64 and 32 channels and a kernel size of 2 with ReLU nonlinearities, followed by maxpooling of size 2 and dropout of 0.3. The convolutional block is followed by a fully connected layer of 256 with ReLU nonlinearity, and finally a fully connected layer is used to predict the class logits (10 in MNIST case).

__init__(*output_dim=10*, ****kwargs**)

Constructor.

Parameters

output_dim (int) – Output dimension

call(*x*, *training=True*, ****kwargs**)

Forward pass.

Parameters

- **x** (Tensor) – Input tensor.
- **training** (bool) – Training flag.
- ****kwargs** – Other arguments. Not used.

Return type

Tensor

Returns

Classification logits.

class alibi.models.tensorflow.cfml_models.**MNISTDecoder**(****kwargs**)

Bases: Model

MNIST decoder used in the Counterfactual with Reinforcement Learning experiments. The model consists of a fully connected layer of 128 units with ReLU activation followed by a convolutional block. The convolutional block consists of 4 convolutional layers having 8, 8, 8 and 1 channels and a kernel size of 3. Each convolutional layer, except the last one, has ReLU nonlinearities and is followed by an up-sampling layer of size 2. The final layers uses a sigmoid activation to clip the output values in [0, 1].

__init__(****kwargs**)

Constructor.

call(*x*, ****kwargs**)

Forward pass.

Parameters

- **x** (Tensor) – Input tensor
- ****kwargs** – Other arguments. Not used.

Return type
Tensor

Returns
Decoded input having each component in the interval [0, 1].

class alibi.models.tensorflow.cfrl_models.**MNISTEncoder**(latent_dim, **kwargs)

Bases: Model

MNIST encoder used in the experiments for the Counterfactual with Reinforcement Learning. The model consists of 3 convolutional layers having 16, 8 and 8 channels and a kernel size of 3, with ReLU nonlinearities. Each convolutional layer is followed by a maxpooling layer of size 2. Finally, a fully connected layer follows the convolutional block with a tanh nonlinearity. The tanh clips the output between [-1, 1], required in the DDPG algorithm (e.g., [act_low, act_high]). The embedding dimension used in the paper is 32, although this can vary.

__init__(latent_dim, **kwargs)

Constructor.

Parameters
latent_dim (int) – Latent dimension.

call(x, **kwargs)

Forward pass.

Parameters

- **x** (Tensor) – Input tensor.
- ****kwargs** – Other arguments. Not used.

Return type
Tensor

Returns
Encoding representation having each component in the interval [-1, 1]

alibi.prototypes package

The ‘alibi.prototypes’ modules includes prototypes and criticism selection methods.

class alibi.prototypes.**ProtoSelect**(kernel_distance, eps, lambda_penalty=None, batch_size=10000000000, preprocess_fn=None, verbose=False)

Bases: *Summariser*, *FitMixin*

__init__(kernel_distance, eps, lambda_penalty=None, batch_size=10000000000, preprocess_fn=None, verbose=False)

Prototype selection for dataset distillation and interpretable classification proposed by Bien and Tibshirani (2012): <https://arxiv.org/abs/1202.5933>

Parameters

- **kernel_distance** (Callable[[ndarray, ndarray], ndarray]) – Kernel distance to be used. Expected to support computation in batches. Given an input x of size $N \times x_{f1} \times x_{f2} \times \dots$ and an input y of size $N_y \times x_{f1} \times x_{f2} \times \dots$, the kernel distance should return a kernel matrix of size $N \times N_y$.
- **eps** (float) – Epsilon ball size.

- **lambda_penalty** (Optional[float]) – Penalty for each prototype. Encourages a lower number of prototypes to be selected. Corresponds to λ in the paper notation. If not specified, the default value is set to $1/N$ where N is the size of the dataset to choose the prototype instances from, passed to the `alibi.prototypes.prototype.select.ProtoSelect.fit()` method.
- **batch_size** (int) – Batch size to be used for kernel matrix computation.
- **preprocess_fn** (Optional[Callable[[Union[list, ndarray]], ndarray]]) – Preprocessing function used for kernel matrix computation. The preprocessing function takes the input as a *list* or a *numpy* array and transforms it into a *numpy* array which is then fed to the `kernel_distance` function. The use of `preprocess_fn` allows the method to be applied to any data modality.
- **verbose** (bool) – Whether to display progression bar while computing prototype points.

fit(*X*, *y=None*, *Z=None*)

Fit the summariser. This step forms the kernel matrix in memory which has a shape of $NX \times NX$, where NX is the number of instances in *X*, if the optional dataset *Z* is not provided. Otherwise, if the optional dataset *Z* is provided, the kernel matrix has a shape of $NZ \times NX$, where NZ is the number of instances in *Z*.

Parameters

- **X** (Union[list, ndarray]) – Dataset to be summarised.
- **y** (Optional[ndarray]) – Labels of the dataset *X* to be summarised. The labels are expected to be represented as integers $[0, 1, \dots, L-1]$, where L is the number of classes in the dataset *X*.
- **Z** (Union[list, ndarray, None]) – Optional dataset to choose the prototypes from. If *Z=None*, the prototypes will be selected from the dataset *X*. Otherwise, if *Z* is provided, the dataset to be summarised is still *X*, but it is summarised by prototypes belonging to the dataset *Z*.

Return type

ProtoSelect

Returns

self – Reference to itself.

summarise(*num_prototypes=1*)

Searches for the requested number of prototypes. Note that the algorithm can return a lower number of prototypes than the requested one. To increase the number of prototypes, reduce the epsilon-ball radius (*eps*), and the penalty for adding a prototype (*lambda_penalty*).

Parameters

num_prototypes (int) – Maximum number of prototypes to be selected.

Return type

Explanation

Returns

An *Explanation* object containing the prototypes, prototype indices and prototype labels with additional metadata as attributes.

alibi.prototypes.visualize_image_prototypes(*summary*, *trainset*, *reducer*, *preprocess_fn=None*, *knn_kw=None*, *ax=None*, *fig_kw=None*, *image_size=(28, 28)*, *zoom_lb=1.0*, *zoom_ub=3.0*)

Plot the images of the prototypes at the location given by the *reducer* representation. The size of each prototype is proportional to the logarithm of the number of assigned training instances correctly classified according to the 1-KNN classifier (Bien and Tibshirani (2012): <https://arxiv.org/abs/1202.5933>).

Parameters

- **summary** (*Explanation*) – An *Explanation* object produced by a call to the `alibi.prototypes.protoselect.ProtoSelect.summarise()` method.
- **trainset** (`Tuple[ndarray, ndarray]`) – Tuple, (X_{train} , y_{train}), consisting of the training data instances with the corresponding labels.
- **reducer** (`Callable[[ndarray], ndarray]`) – 2D reducer. Reduces the input feature representation to 2D. Note that the reducer operates directly on the input instances if `preprocess_fn=None`. If the `preprocess_fn` is specified, the reducer will be called on the feature representation obtained after passing the input instances through the `preprocess_fn`.
- **preprocess_fn** (`Optional[Callable[[ndarray], ndarray]]`) – Optional preprocessor function. If `preprocess_fn=None`, no preprocessing is applied.
- **knn_kw** (`Optional[dict]`) – Keyword arguments passed to `sklearn.neighbors.KNeighborsClassifier`. The `n_neighbors` will be set automatically to 1, but the `metric` has to be specified according to the kernel distance used. If the `metric` is not specified, it will be set by default to 'euclidean'. See parameters description: <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>
- **ax** (`Optional[Axes]`) – A `matplotlib` axes object to plot on.
- **fig_kw** (`Optional[dict]`) – Keyword arguments passed to the `fig.set` function.
- **image_size** (`Tuple[int, int]`) – Shape to which the prototype images will be resized. A zoom of 1 will display the image having the shape `image_size`.
- **zoom_lb** (`float`) – Zoom lower bound. The zoom will be scaled linearly between `[zoom_lb, zoom_ub]`.
- **zoom_ub** (`float`) – Zoom upper bound. The zoom will be scaled linearly between `[zoom_lb, zoom_ub]`.

Return type

Axes

Submodules

alibi.prototypes.protoselect module

```
class alibi.prototypes.protoselect.ProtoSelect(kernel_distance, eps, lambda_penalty=None,
                                              batch_size=10000000000, preprocess_fn=None,
                                              verbose=False)
```

Bases: `Summariser`, `FitMixin`

```
__init__(kernel_distance, eps, lambda_penalty=None, batch_size=10000000000, preprocess_fn=None,
         verbose=False)
```

Prototype selection for dataset distillation and interpretable classification proposed by Bien and Tibshirani (2012): <https://arxiv.org/abs/1202.5933>

Parameters

- **kernel_distance** (`Callable[[ndarray, ndarray], ndarray]`) – Kernel distance to be used. Expected to support computation in batches. Given an input x of size $N_x \times f_1 \times f_2 \times \dots$ and an input y of size $N_y \times f_1 \times f_2 \times \dots$, the kernel distance should return a kernel matrix of size $N_x \times N_y$.

- **eps** (`float`) – Epsilon ball size.
- **lambda_penalty** (`Optional[float]`) – Penalty for each prototype. Encourages a lower number of prototypes to be selected. Corresponds to λ in the paper notation. If not specified, the default value is set to $1/N$ where N is the size of the dataset to choose the prototype instances from, passed to the `alibi.prototypes.protoselect.ProtoSelect.fit()` method.
- **batch_size** (`int`) – Batch size to be used for kernel matrix computation.
- **preprocess_fn** (`Optional[Callable[[Union[list, ndarray], ndarray]]`) – Preprocessing function used for kernel matrix computation. The preprocessing function takes the input as a *list* or a *numpy* array and transforms it into a *numpy* array which is then fed to the `kernel_distance` function. The use of `preprocess_fn` allows the method to be applied to any data modality.
- **verbose** (`bool`) – Whether to display progression bar while computing prototype points.

fit(*X*, *y=None*, *Z=None*)

Fit the summariser. This step forms the kernel matrix in memory which has a shape of $NX \times NX$, where NX is the number of instances in *X*, if the optional dataset *Z* is not provided. Otherwise, if the optional dataset *Z* is provided, the kernel matrix has a shape of $NZ \times NX$, where NZ is the number of instances in *Z*.

Parameters

- **X** (`Union[list, ndarray]`) – Dataset to be summarised.
- **y** (`Optional[ndarray]`) – Labels of the dataset *X* to be summarised. The labels are expected to be represented as integers $[0, 1, \dots, L-1]$, where L is the number of classes in the dataset *X*.
- **Z** (`Union[list, ndarray, None]`) – Optional dataset to choose the prototypes from. If *Z=None*, the prototypes will be selected from the dataset *X*. Otherwise, if *Z* is provided, the dataset to be summarised is still *X*, but it is summarised by prototypes belonging to the dataset *Z*.

Return type

`ProtoSelect`

Returns

self – Reference to itself.

summarise(*num_prototypes=1*)

Searches for the requested number of prototypes. Note that the algorithm can return a lower number of prototypes than the requested one. To increase the number of prototypes, reduce the epsilon-ball radius (*eps*), and the penalty for adding a prototype (*lambda_penalty*).

Parameters

num_prototypes (`int`) – Maximum number of prototypes to be selected.

Return type

`Explanation`

Returns

An *Explanation* object containing the prototypes, prototype indices and prototype labels with additional metadata as attributes.

`alibi.prototypes.protoselect.compute_prototype_importances(summary, trainset,`
`preprocess_fn=None, knn_kw=None)`

Computes the importance of each prototype. The importance of a prototype is the number of assigned training

instances correctly classified according to the 1-KNN classifier (Bien and Tibshirani (2012): <https://arxiv.org/abs/1202.5933>).

Parameters

- **summary** (*Explanation*) – An *Explanation* object produced by a call to the `alibi.prototypes.protoselect.ProtoSelect.summarise()` method.
- **trainset** (`Tuple[ndarray, ndarray]`) – Tuple, (*X_train*, *y_train*), consisting of the training data instances with the corresponding labels.
- **preprocess_fn** (`Optional[Callable[[ndarray], ndarray]]`) – Optional preprocessor function. If `preprocess_fn=None`, no preprocessing is applied.
- **knn_kw** (`Optional[dict]`) – Keyword arguments passed to `sklearn.neighbors.KNeighborsClassifier`. The *n_neighbors* will be set automatically to 1, but the *metric* has to be specified according to the kernel distance used. If the *metric* is not specified, it will be set by default to 'euclidean'. See parameters description: <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>

Return type

`Dict[str, Optional[ndarray]]`

Returns

A dictionary containing –

- 'prototype_indices' - an array of the prototype indices.
- 'prototype_importances' - an array of prototype importances.
- 'X_protos' - an array of raw prototypes.
- 'X_protos_ft' - an optional array of preprocessed prototypes. If the `preprocess_fn=None`, no preprocessing is applied and `None` is returned instead.

```
alibi.prototypes.protoselect.cv_protoselect_euclidean(trainset, protoset=None, valset=None,
                                                    num_prototypes=1, eps_grid=None,
                                                    quantiles=None, grid_size=25, n_splits=2,
                                                    batch_size=10000000000,
                                                    preprocess_fn=None, protoselect_kw=None,
                                                    knn_kw=None, kfold_kw=None)
```

Cross-validation parameter selection for *ProtoSelect* with Euclidean distance. The method computes the best epsilon radius.

Parameters

- **trainset** (`Tuple[ndarray, ndarray]`) – Tuple, (*X_train*, *y_train*), consisting of the training data instances with the corresponding labels.
- **protoset** (`Optional[Tuple[ndarray]]`) – Tuple, (*Z*), consisting of the dataset to choose the prototypes from. If *Z* is not provided (i.e., `protoset=None`), the prototypes will be selected from the training dataset *X*. Otherwise, if *Z* is provided, the dataset to be summarised is still *X*, but it is summarised by prototypes belonging to the dataset *Z*. Note that the argument is passed as a tuple with a single element for consistency reasons.
- **valset** (`Optional[Tuple[ndarray, ndarray]]`) – Optional tuple (*X_val*, *y_val*) consisting of validation data instances with the corresponding validation labels. 1-KNN classifier is evaluated on the validation dataset to obtain the best epsilon radius. In case `valset=None`, then *n_splits* cross-validation is performed on the *trainset*.
- **num_prototypes** (`int`) – The number of prototypes to be selected.

- **eps_grid** (`Optional[ndarray]`) – Optional grid of values to select the epsilon radius from. If not specified, the search grid is automatically proposed based on the inter-distances between X and Z . The distances are filtered by considering only values in between the *quantiles* values. The minimum and maximum distance values are used to define the range of values to search the epsilon radius. The interval is discretized in *grid_size* equidistant bins.
- **quantiles** (`Optional[Tuple[float, float]]`) – Quantiles, (q_{min} , q_{max}), to be used to filter the range of values of the epsilon radius. The expected quantile values are in $[0, 1]$ and clipped to $[0, 1]$ if outside the range. See *eps_grid* for usage. If not specified, no filtering is applied. Only used if *eps_grid*=None.
- **grid_size** (`int`) – The number of equidistant bins to be used to discretize the *eps_grid* automatically proposed interval. Only used if *eps_grid*=None.
- **batch_size** (`int`) – Batch size to be used for kernel matrix computation.
- **preprocess_fn** (`Optional[Callable[[ndarray], ndarray]]`) – Preprocessing function to be applied to the data instance before applying the kernel.
- **protoselect_kw** (`Optional[dict]`) – Keyword arguments passed to `alibi.prototypes.protoselect.ProtoSelect.__init__()`.
- **knn_kw** (`Optional[dict]`) – Keyword arguments passed to `sklearn.neighbors.KNeighborsClassifier`. The *n_neighbors* will be set automatically to 1 and the *metric* will be set to 'euclidean'. See parameters description: <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>
- **kfold_kw** (`Optional[dict]`) – Keyword arguments passed to `sklearn.model_selection.KFold`. See parameters description: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

Return type`dict`**Returns**

Dictionary containing –

- 'best_eps': float - the best epsilon radius according to the accuracy of a 1-KNN classifier.
- 'meta': dict - dictionary containing argument and data gather throughout cross-validation.

```
alibi.prototypes.protoselect.visualize_image_prototypes(summary, trainset, reducer,  
                                                         preprocess_fn=None, knn_kw=None,  
                                                         ax=None, fig_kw=None, image_size=(28,  
                                                         28), zoom_lb=1.0, zoom_ub=3.0)
```

Plot the images of the prototypes at the location given by the *reducer* representation. The size of each prototype is proportional to the logarithm of the number of assigned training instances correctly classified according to the 1-KNN classifier (Bien and Tibshirani (2012): <https://arxiv.org/abs/1202.5933>).

Parameters

- **summary** (*Explanation*) – An *Explanation* object produced by a call to the `alibi.prototypes.protoselect.ProtoSelect.summarise()` method.
- **trainset** (`Tuple[ndarray, ndarray]`) – Tuple, (X_{train} , y_{train}), consisting of the training data instances with the corresponding labels.
- **reducer** (`Callable[[ndarray], ndarray]`) – 2D reducer. Reduces the input feature representation to 2D. Note that the reducer operates directly on the input instances if

`preprocess_fn=None`. If the `preprocess_fn` is specified, the reducer will be called on the feature representation obtained after passing the input instances through the `preprocess_fn`.

- **`preprocess_fn`** (`Optional[Callable[[ndarray], ndarray]]`) – Optional preprocessor function. If `preprocess_fn=None`, no preprocessing is applied.
- **`knn_kw`** (`Optional[dict]`) – Keyword arguments passed to `sklearn.neighbors.KNeighborsClassifier`. The `n_neighbors` will be set automatically to 1, but the `metric` has to be specified according to the kernel distance used. If the `metric` is not specified, it will be set by default to 'euclidean'. See parameters description: <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>
- **`ax`** (`Optional[Axes]`) – A `matplotlib` axes object to plot on.
- **`fig_kw`** (`Optional[dict]`) – Keyword arguments passed to the `fig.set` function.
- **`image_size`** (`Tuple[int, int]`) – Shape to which the prototype images will be resized. A zoom of 1 will display the image having the shape `image_size`.
- **`zoom_lb`** (`float`) – Zoom lower bound. The zoom will be scaled linearly between `[zoom_lb, zoom_ub]`.
- **`zoom_ub`** (`float`) – Zoom upper bound. The zoom will be scaled linearly between `[zoom_lb, zoom_ub]`.

Return type

Axes

alibi.tests package

Submodules

alibi.tests.utils module

class `alibi.tests.utils.MockPredictor(out_dim, out_type='proba', model_type=None, seed=None)`

Bases: `object`

A class that mimicks the output of a classifier or regressor to allow testing of functionality that depends on it without inference overhead.

`__init__` (`out_dim, out_type='proba', model_type=None, seed=None`)

Parameters

- **`out_dim`** (`int`) – The number of output classes.
- **`out_type`** (`str`) – Indicates if probabilities, class predictions or continuous outputs are generated.

`predict` (`*args, **kwargs`)

`alibi.tests.utils.assert_message_in_logs(msg, records)`

Helper function to check if a msg is present in any of the records (an iterable of strings).

`alibi.tests.utils.issorted(arr, reverse=False)`

Checks if a numpy array is sorted.

`alibi.tests.utils.not_raises(ExpectedException)`

A context manager used to check that `ExpectedException` does not occur during testing.

alibi.utils package

class `alibi.utils.BertBaseUncased`(*preloading=True*)

Bases: [`LanguageModel`](#)

SUBWORD_PREFIX = '##'

Language model subword prefix.

__init__(*preloading=True*)

Initialize *BertBaseUncased*.

Parameters

preloading (`bool`) – See [`alibi.utils.lang_model.LanguageModel.__init__\(\)`](#).

is_subword_prefix(*token*)

Checks if the given token is a part of the tail of a word. Note that a word can be split in multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). Each language model has a convention on how to mark a tail token. For example *DistilbertBaseUncased* and *BertBaseUncased* have the tail tokens prefixed with the special set of characters '##'. On the other hand, for *RobertaBase* only the head token is prefixed with the special character 'Ġ' and thus we need to check the absence of the prefix to identify the tail tokens. We call those special characters *SUBWORD_PREFIX*. Due to different conventions, this method has to be implemented for each language model. See module docstring for namings.

Parameters

token (`str`) – Token to be checked if it is a subword.

Return type

`bool`

Returns

True if the given token is a subword prefix. False otherwise.

property mask: `str`

Returns the mask token.

Return type

`str`

class `alibi.utils.DistilbertBaseUncased`(*preloading=True*)

Bases: [`LanguageModel`](#)

SUBWORD_PREFIX = '##'

Language model subword prefix.

__init__(*preloading=True*)

Initialize *DistilbertBaseUncased*.

Parameters

preloading (`bool`) – See [`alibi.utils.lang_model.LanguageModel.__init__\(\)`](#).

is_subword_prefix(*token*)

Checks if the given token is a part of the tail of a word. Note that a word can be split in multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). Each language model has a convention on how to mark a tail token. For example *DistilbertBaseUncased* and *BertBaseUncased* have the tail tokens prefixed with the special set of characters '##'. On the other hand, for *RobertaBase* only the head token is prefixed with the special character 'Ġ' and thus we need to check the absence of the prefix to identify the tail tokens. We call those special characters *SUBWORD_PREFIX*. Due to

different conventions, this method has to be implemented for each language model. See module docstring for namings.

Parameters

token (`str`) – Token to be checked if it is a subword.

Return type

`bool`

Returns

True if the given token is a subword prefix. False otherwise.

property mask: `str`

Returns the mask token.

Return type

`str`

```
class alibi.utils.DistributedExplainer(distributed_opts, explainer_type, explainer_init_args,
                                     explainer_init_kwargs, concatenate_results=True,
                                     return_generator=False)
```

Bases: `object`

A class that orchestrates the execution of the execution of a batch of explanations in parallel.

__getattr__ (*item*)

Accesses actor attributes. Use sparingly as this involves a remote call (that is, these attributes are of an object in a different process). *The intended use is for retrieving any common state across the actor at the end of the computation in order to form the response (see notes 2 & 3).*

Parameters

item (`str`) – The explainer attribute to be returned.

Return type

`Any`

Returns

The value of the attribute specified by *item*.

Raises

ValueError – If the actor index is invalid.

Notes

1. This method assumes that the actor implements a *return_attribute* method.
2. Note that we are indexing the idle actors. This means that if a pool was initialised with 5 actors and 3 are busy, indexing with index 2 will raise an *IndexError*.
3. The order of *_idle_actors* constantly changes - an actor is removed from it if there is a task to execute and appended back when the task is complete. Therefore, indexing at the same position as computation proceeds will result in retrieving state from different processes.

```
__init__ (distributed_opts, explainer_type, explainer_init_args, explainer_init_kwargs,
          concatenate_results=True, return_generator=False)
```

Creates a pool of actors (i.e., replicas of an instantiated *explainer_type* in a separate process) which can explain batches of instances in parallel via calls to *get_explanation*.

Parameters

- **distributed_opts** (`Dict[str, Any]`) – A dictionary with the following type (minimal signature):

```
class DistributedOpts(TypedDict):  
    n_cpus: Optional[int]  
    batch_size: Optional[int]
```

The dictionary may contain two additional keys:

- `'actor_cpu_frac'` : (`float`, `<= 1.0`, `>0.0`) - This is used to create more than one process on one CPU/GPU. This may not speed up CPU intensive tasks but it is worth experimenting with when few physical cores are available. In particular, this is highly useful when the user wants to share a GPU for multiple tasks, with the caveat that the machine learning framework itself needs to support running multiple replicas on the same GPU. See the *ray* documentation [here](#) for details.
 - `'algorithm'` : `str` - this is specified internally by the caller. It is used in order to register target function callbacks for the parallel pool. These should be implemented in the global scope. If not specified, its value will be `'default'`, which will select a default target function which expects the actor has a `get_explanation` method.
- **explainer_type** (`Any`) – Explainer class.
 - **explainer_init_args** (`Tuple`) – Positional arguments to explainer constructor.
 - **explainer_init_kwargs** (`dict`) – Keyword arguments to explainer constructor.
 - **concatenate_results** (`bool`) – If `True` concatenates the results. See [alibi.utils.distributed.concatenate_minibatches\(\)](#) for more details.
 - **return_generator** (`bool`) – If `True` a generator that returns the results in the order the computation finishes is returned when `get_explanation` is called. Otherwise, the order of the results is the same as the order of the minibatches.

Notes

When `return_generator=True`, the caller has to take elements from the generator (e.g., by calling `next`) in order to start computing the results (because the *ray* pool is implemented as a generator).

property `actor_index`: `int`

Returns the index of the actor for which state is returned.

Return type

`int`

concatenate: `Callable`

create_parallel_pool(`explainer_type`, `explainer_init_args`, `explainer_init_kwargs`)

Creates a pool of actors that can explain the rows of a dataset in parallel.

Parameters

documentation. (See *constructor*) –

get_explanation(`X`, `**kwargs`)

Performs distributed explanations of instances in `X`.

Parameters

- **X** (ndarray) – A batch of instances to be explained. Split into batches according to the settings passed to the constructor.
- ****kwargs** – Any keyword-arguments for the explainer *explain* method.

Return type

`Union[Generator[Tuple[int, Any], None, None], List[Any], Any]`

Returns

The explanations are returned as –

- a generator, if the *return_generator* option is specified. This is used so that the caller can access the results as they are computed. This is the only case when this method is non-blocking and the caller needs to call *next* on the generator to trigger the parallel computation.
- a list of objects, whose type depends on the return type of the explainer. This is returned if no custom preprocessing function is specified.
- an object, whose type depends on the return type of the concatenation function return when called with a list of minibatch results with the same order as the minibatches.

return_attribute(*name*)

Returns an attribute specified by its name. Used in a distributed context where the properties cannot be accessed using the dot syntax.

Return type

`Any`

set_actor_index(*value*)

Sets actor index. This is used when the *DistributedExplainer* is in a separate process because *ray* does not support calling property setters remotely

class `alibi.utils.LanguageModel`(*model_path*, *preloading=True*)

Bases: `ABC`

SUBWORD_PREFIX = ''

Language model subword prefix.

__init__(*model_path*, *preloading=True*)

Initialize the language model.

Parameters

- **model_path** (`str`) – *transformers* package model path.
- **preloading** (`bool`) – Whether to preload the online version of the transformer. If `False`, a call to *from_disk* method is expected.

caller: `Callable`

from_disk(*path*)

Loads a model from disk.

Parameters

path (`Union[str, Path]`) – Path to the checkpoint.

head_tail_split(*text*)

Split the text in head and tail. Some language models support a maximum number of tokens. Thus is necessary to split the text to meet this constraint. After the text is split in head and tail, only the head is considered for operation. Thus the tail will remain unchanged.

Parameters

text (`str`) – Text to be split in head and tail.

Return type

`Tuple[str, str, List[str], List[str]]`

Returns

Tuple consisting of the head, tail and their corresponding list of tokens.

is_punctuation(*token, punctuation*)

Checks if the given token is punctuation.

Parameters

- **token** (`str`) – Token to be checked if it is punctuation.
- **punctuation** (`str`) – String containing all punctuation to be considered.

Return type

`bool`

Returns

True if the *token* is a punctuation. False otherwise.

is_stop_word(*tokenized_text, start_idx, punctuation, stopwords*)

Checks if the given word starting at the given index is in the list of stopwords.

Parameters

- **tokenized_text** (`List[str]`) – Tokenized text.
- **start_idx** (`int`) – Starting index of a word.
- **stopwords** (`Optional[List[str]]`) – List of stop words. The words in this list should be lowercase.
- **punctuation** (`str`) – Punctuation to be considered. See `alibi.utils.lang_model.LanguageModel.select_entire_word()`.

Return type

`bool`

Returns

True if the *token* is in the *stopwords* list. False otherwise.

abstract is_subword_prefix(*token*)

Checks if the given token is a part of the tail of a word. Note that a word can be split in multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). Each language model has a convention on how to mark a tail token. For example *DistilbertBaseUncased* and *BertBaseUncased* have the tail tokens prefixed with the special set of characters '##'. On the other hand, for *RobertaBase* only the head token is prefixed with the special character 'Ġ' and thus we need to check the absence of the prefix to identify the tail tokens. We call those special characters *SUBWORD_PREFIX*. Due to different conventions, this method has to be implemented for each language model. See module docstring for namings.

Parameters

token (`str`) – Token to be checked if it is a subword.

Return type

`bool`

Returns

True if the given token is a subword prefix. False otherwise.

abstract property mask: `str`

Returns the mask token.

Return type

`str`

property mask_id: `int`

Returns the mask token id

Return type

`int`

property max_num_tokens: `int`

Returns the maximum number of token allowed by the model.

Return type

`int`

model: `Any`

predict_batch_lm(*x*, *vocab_size*, *batch_size*)

Tensorflow language model batch predictions for *AnchorText*.

Parameters

- **x** (`BatchEncoding`) – Batch of instances.
- **vocab_size** (`int`) – Vocabulary size of language model.
- **batch_size** (`int`) – Batch size used for predictions.

Return type

`ndarray`

Returns

y – Array with model predictions.

select_word(*tokenized_text*, *start_idx*, *punctuation*)

Given a tokenized text and the starting index of a word, the function selects the entire word. Note that a word is composed of multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). The tail tokens can be identified based on the presence/absence of `SUBWORD_PREFIX`. See [alibi.utils.lang_model.LanguageModel.is_subword_prefix\(\)](#) for more details.

Parameters

- **tokenized_text** (`List[str]`) – Tokenized text.
- **start_idx** (`int`) – Starting index of a word.
- **punctuation** (`str`) – String of punctuation to be considered. If it encounters a token composed only of characters in *punctuation* it terminates the search.

Return type

`str`

Returns

The word obtained by concatenation `[head_token tail_token_1 tail_token_2 ... tail_token_k]`.

to_disk(*path*)

Saves a model to disk.

Parameters

path (`Union[str, Path]`) – Path to the checkpoint.

tokenizer: `Any`

class `alibi.utils.RobertaBase(preloading=True)`

Bases: `LanguageModel`

SUBWORD_PREFIX = `'Ġ'`

Language model subword prefix.

__init__ (`preloading=True`)

Initialize *RobertaBase*.

Parameters

preloading (`bool`) – See `alibi.utils.lang_model.LanguageModel.__init__()` constructor.

is_subword_prefix (`token`)

Checks if the given token is a part of the tail of a word. Note that a word can be split in multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). Each language model has a convention on how to mark a tail token. For example *DistilbertBaseUncased* and *BertBaseUncased* have the tail tokens prefixed with the special set of characters `'##'`. On the other hand, for *RobertaBase* only the head token is prefixed with the special character `'Ġ'` and thus we need to check the absence of the prefix to identify the tail tokens. We call those special characters *SUBWORD_PREFIX*. Due to different conventions, this method has to be implemented for each language model. See module docstring for namings.

Parameters

token (`str`) – Token to be checked if it is a subword.

Return type

`bool`

Returns

True if the given token is a subword prefix. False otherwise.

property mask: `str`

Returns the mask token.

Return type

`str`

`alibi.utils.gen_category_map(data, categorical_columns=None)`

Parameters

- **data** (`Union[DataFrame, ndarray]`) – 2-dimensional *pandas* dataframe or *numpy* array.
- **categorical_columns** (`Union[List[int], List[str], None]`) – A list of columns indicating categorical variables. Optional if passing a *pandas* dataframe as inference will be used based on dtype `'O'`. If passing a *numpy* array this is compulsory.

Return type

`Dict[int, list]`

Returns

category_map – A dictionary with keys being the indices of the categorical columns and values being lists of categories for that column. Implicitly each category is mapped to the index of its position in the list.

`alibi.utils.ohe_to_ord(X_ohe, cat_vars_ohe)`

Convert one-hot encoded variables to ordinal encodings.

Parameters

- **X_ohe** (ndarray) – Data with mixture of one-hot encoded and numerical variables.
- **cat_vars_ohe** (dict) – Dict with as keys the first column index for each one-hot encoded categorical variable and as values the number of categories per categorical variable.

Return type

`Tuple[ndarray, dict]`

Returns

Ordinal equivalent of one-hot encoded data and dict with categorical columns and number of categories.

`alibi.utils.ord_to_ohe(X_ord, cat_vars_ord)`

Convert ordinal to one-hot encoded variables.

Parameters

- **X_ord** (ndarray) – Data with mixture of ordinal encoded and numerical variables.
- **cat_vars_ord** (dict) – Dict with as keys the categorical columns and as values the number of categories per categorical variable.

Return type

`Tuple[ndarray, dict]`

Returns

One-hot equivalent of ordinal encoded data and dict with categorical columns and number of categories.

`alibi.utils.spacy_model(model='en_core_web_md')`

Download spaCy model.

Parameters

model (str) – Model to be downloaded.

Return type

`None`

`alibi.utils.visualize_image_attr(attr, original_image=None, method='heat_map', sign='absolute_value',
plt_fig_axis=None, outlier_perc=2, cmap=None, alpha_overlay=0.5,
show_colorbar=False, title=None, fig_size=(6, 6), use_pyplot=True)`

Visualizes attribution for a given image by normalizing attribution values of the desired sign ('positive' | 'negative' | 'absolute_value' | 'all') and displaying them using the desired mode in a *matplotlib* figure.

Parameters

- **attr** (ndarray) – *Numpy* array corresponding to attributions to be visualized. Shape must be in the form (*H*, *W*, *C*), with channels as last dimension. Shape must also match that of the original image if provided.
- **original_image** (Optional[ndarray]) – *Numpy* array corresponding to original image. Shape must be in the form (*H*, *W*, *C*), with channels as the last dimension. Image can be provided either with *float* values in range 0-1 or *int* values between 0-255. This is a necessary argument for any visualization method which utilizes the original image.
- **method** (str) – Chosen method for visualizing attribution. Supported options are:
 - 'heat_map' - Display heat map of chosen attributions

- 'blended_heat_map' - Overlay heat map over greyscale version of original image. Parameter `alpha_overlay` corresponds to alpha of heat map.
- 'original_image' - Only display original image.
- 'masked_image' - Mask image (pixel-wise multiply) by normalized attribution values.
- 'alpha_scaling' - Sets alpha channel of each pixel to be equal to normalized attribution value.

Default: 'heat_map'.

- **sign** (`str`) – Chosen sign of attributions to visualize. Supported options are:
 - 'positive' - Displays only positive pixel attributions.
 - 'absolute_value' - Displays absolute value of attributions.
 - 'negative' - Displays only negative pixel attributions.
 - 'all' - Displays both positive and negative attribution values. This is not supported for 'masked_image' or 'alpha_scaling' modes, since signed information cannot be represented in these modes.
- **plt_fig_axis** (`Optional[Tuple[Figure, Axes]]`) – Tuple of `matplotlib.pyplot.figure` and `axis` on which to visualize. If `None` is provided, then a new figure and axis are created.
- **outlier_perc** (`Union[int, float]`) – Top attribution values which correspond to a total of `outlier_perc` percentage of the total attribution are set to 1 and scaling is performed using the minimum of these values. For `sign='all'`, outliers and scale value are computed using absolute value of attributions.
- **cmap** (`Optional[str]`) – String corresponding to desired colormap for heatmap visualization. This defaults to 'Reds' for negative sign, 'Blues' for absolute value, 'Greens' for positive sign, and a spectrum from red to green for all. Note that this argument is only used for visualizations displaying heatmaps.
- **alpha_overlay** (`float`) – Visualizes attribution for a given image by normalizing attribution values of the desired sign (positive, negative, absolute value, or all) and displaying them using the desired mode in a matplotlib figure.
- **show_colorbar** (`bool`) – Displays colorbar for heatmap below the visualization. If given method does not use a heatmap, then a colormap axis is created and hidden. This is necessary for appropriate alignment when visualizing multiple plots, some with colorbars and some without.
- **title** (`Optional[str]`) – The title for the plot. If `None`, no title is set.
- **fig_size** (`Tuple[int, int]`) – Size of figure created.
- **use_pyplot** (`bool`) – If `True`, uses `pyplot` to create and show figure and displays the figure after creating. If `False`, uses `matplotlib` object-oriented API and simply returns a figure object without showing.

Return type

`Tuple[Figure, Axes]`

Returns

2-element tuple of consisting of –

- `figure` : `matplotlib.pyplot.Figure` - Figure object on which visualization is created. If `plt_fig_axis` argument is given, this is the same figure provided.

- *axis* : `matplotlib.pyplot.Axes` - Axes object on which visualization is created. If *plt_fig_axis* argument is given, this is the same axis provided.

Submodules

alibi.utils.approximation_methods module

class `alibi.utils.approximation_methods.Riemann(value)`

Bases: `Enum`

An enumeration.

left = 1

middle = 3

right = 2

trapezoid = 4

`alibi.utils.approximation_methods.SUPPORTED_RIEMANN_METHODS` = ['riemann_left', 'riemann_right', 'riemann_middle', 'riemann_trapezoid']

Riemann integration methods.

`alibi.utils.approximation_methods.approximation_parameters(method)`

Retrieves parameters for the input approximation *method*.

Parameters

method (`str`) – The name of the approximation method. Currently supported only: 'riemann_*' and 'gausslegendre'. Check `alibi.utils.approximation_methods.SUPPORTED_RIEMANN_METHODS` for all 'riemann_*' possible values.

Return type

`Tuple[Callable[[int], List[float]], Callable[[int], List[float]]]`

`alibi.utils.approximation_methods.gauss_legendre_builders()`

`np.polynomial.legendre` function helps to compute step sizes and alpha coefficients using gauss-legendre quadrature rule. Since `numpy` returns the integration parameters in different scales we need to rescale them to adjust to the desired scale.

Gauss Legendre quadrature rule for approximating the integrals was originally proposed by [Xue Feng and her intern Hauroun Habeeb] (<https://research.fb.com/people/feng-xue/>).

Parameters

n – The number of integration steps.

Return type

`Tuple[Callable[[int], List[float]], Callable[[int], List[float]]]`

Returns

2-element tuple consisting of –

- *step_sizes* : `Callable` - *step_sizes* takes the number of steps as an input argument and returns an array of steps sizes which sum is smaller than or equal to one.
- *alphas* : `Callable` - *alphas* takes the number of steps as an input argument and returns the multipliers/coefficients for the inputs of integrand in the range of [0, 1].

`alibi.utils.approximation_methods.riemann_builders(method=Riemann.trapezoid)`

Step sizes are identical and alphas are scaled in [0, 1].

Parameters

- **n** – The number of integration steps.
- **method** (*Riemann*) – Riemann method: `Riemann.left` | `Riemann.right` | `Riemann.middle` | `Riemann.trapezoid`.

Return type

`Tuple[Callable[[int], List[float]], Callable[[int], List[float]]]`

Returns

2-element tuple consisting of –

- *step_sizes* : `Callable` - *step_sizes* takes the number of steps as an input argument and returns an array of steps sizes which sum is smaller than or equal to one.
- *alphas* : `Callable` - *alphas* takes the number of steps as an input argument and returns the multipliers/coefficients for the inputs of integrand in the range of [0, 1].

alibi.utils.data module

class `alibi.utils.data.Bunch(**kwargs)`

Bases: `dict`

Container object for internal datasets. Dictionary-like object that exposes its keys as attributes.

`alibi.utils.data.gen_category_map(data, categorical_columns=None)`

Parameters

- **data** (`Union[DataFrame, ndarray]`) – 2-dimensional *pandas* dataframe or *numpy* array.
- **categorical_columns** (`Union[List[int], List[str], None]`) – A list of columns indicating categorical variables. Optional if passing a *pandas* dataframe as inference will be used based on dtype 'O'. If passing a *numpy* array this is compulsory.

Return type

`Dict[int, list]`

Returns

category_map – A dictionary with keys being the indices of the categorical columns and values being lists of categories for that column. Implicitly each category is mapped to the index of its position in the list.

alibi.utils.discretizer module

class `alibi.utils.discretizer.Discretizer(data, numerical_features, feature_names, percentiles=(25, 50, 75))`

Bases: `object`

__init__(`data, numerical_features, feature_names, percentiles=(25, 50, 75)`)

Initialize the discretizer.

Parameters

- **data** (`ndarray`) – Data to discretize.

- **numerical_features** (`List[int]`) – List of indices corresponding to the continuous feature columns. Only these features will be discretized.
- **feature_names** (`List[str]`) – List with feature names.
- **percentiles** (`Sequence[Union[int, float]]`) – Percentiles used for discretization.

bins(*data*)

Parameters

data (ndarray) – Data to discretize.

Return type

`List[ndarray]`

Returns

List with bin values for each feature that is discretized.

discretize(*data*)

Parameters

data (ndarray) – Data to discretize.

Return type

ndarray

Returns

Discretized version of data with the same dimension.

static get_percentiles(*x*, *qts*)

Discretizes the the data in *x* using the quantiles in *qts*. This is achieved by searching for the index of each value in *x* into *qts*, which is assumed to be a 1-D sorted array.

Parameters

- **x** (ndarray) – A *numpy* array of data to be discretized
- **qts** (ndarray) – A *numpy* array of percentiles. This should be a 1-D array sorted in ascending order.

Return type

ndarray

Returns

A discretized data *numpy* array.

alibi.utils.distance module

alibi.utils.distance.abdm(*X*, *cat_vars*, *cat_vars_bin*={})

Calculate the pair-wise distances between categories of a categorical variable using the Association-Based Distance Metric based on Le et al (2005). <http://www.jaist.ac.jp/~bao/papers/N26.pdf>

Parameters

- **X** (ndarray) – Batch of arrays.
- **cat_vars** (`dict`) – Dict with as keys the categorical columns and as optional values the number of categories per categorical variable.
- **cat_vars_bin** (`dict`) – Dict with as keys the binned numerical columns and as optional values the number of bins per variable.

Returns

Dict with as keys the categorical columns and as values the pairwise distance matrix for the variable.

```
alibi.utils.distance.batch_compute_kernel_matrix(x, y, kernel, batch_size=10000000000,
                                                preprocess_fn=None)
```

Compute the kernel matrix between *x* and *y* by filling in blocks of size *batch_size* *x* *batch_size* at a time.

Parameters

- **x** (`Union[list, ndarray]`) – The first list/*numpy* array of data instances.
- **y** (`Union[list, ndarray]`) – The second list/*numpy* array of data instances.
- **kernel** (`Callable[[ndarray, ndarray], ndarray]`) – Kernel function to be used for kernel matrix computation.
- **batch_size** (`int`) – Batch size to be used for each prediction.
- **preprocess_fn** (`Optional[Callable[[Union[list, ndarray]], ndarray]]`) – Optional preprocessing function for each batch.

Return type

`ndarray`

Returns

Kernel matrix in the form of a *numpy* array.

```
alibi.utils.distance.cityblock_batch(X, y)
```

Calculate the L1 distances between a batch of arrays *X* and an array of the same shape *y*.

Parameters

- **X** (`ndarray`) – Batch of arrays to calculate the distances from.
- **y** (`ndarray`) – Array to calculate the distance to.

Return type

`ndarray`

Returns

Array of distances from each array in *X* to *y*.

```
alibi.utils.distance.multidim_scaling(d_pair, feature_range, n_components=2, use_metric=True,
                                     standardize_cat_vars=True, smooth=1.0, center=True,
                                     update_feature_range=True)
```

Apply multidimensional scaling to pairwise distance matrices.

Parameters

- **d_pair** (`dict`) – Dict with as keys the column index of the categorical variables and as values a pairwise distance matrix for the categories of the variable.
- **feature_range** (`Tuple[ndarray, ndarray]`) – Tuple with *min* and *max* ranges to allow for perturbed instances. *Min* and *max* ranges are *numpy* arrays with dimension (1 *x* nb of features).
- **n_components** (`int`) – Number of dimensions in which to immerse the dissimilarities.
- **use_metric** (`bool`) – If True, perform metric MDS; otherwise, perform nonmetric MDS.
- **standardize_cat_vars** (`bool`) – Standardize numerical values of categorical variables if True.

- **smooth** (`float`) – Smoothing exponent between 0 and 1 for the distances. Lower values than 1 will smooth the difference in distance metric between different features.
- **center** (`bool`) – Whether to center the scaled distance measures. If `False`, the min distance for each feature except for the feature with the highest raw max distance will be the lower bound of the feature range, but the upper bound will be below the max feature range.
- **update_feature_range** (`bool`) – Update feature range with scaled values.

Return type`Tuple[dict, tuple]`**Returns***Dict with multidimensional scaled version of pairwise distance matrices.*`alibi.utils.distance.mvdm(X, y, cat_vars, alpha=1)`

Calculate the pair-wise distances between categories of a categorical variable using the Modified Value Difference Measure based on Cost et al (1993). <https://link.springer.com/article/10.1023/A:1022664626993>

Parameters

- **X** (`ndarray`) – Batch of arrays.
- **y** (`ndarray`) – Batch of labels or predictions.
- **cat_vars** (`dict`) – Dict with as keys the categorical columns and as optional values the number of categories per categorical variable.
- **alpha** (`int`) – Power of absolute difference between conditional probabilities.

Return type`Dict[int, ndarray]`**Returns***Dict with as keys the categorical columns and as values the pairwise distance matrix for the variable.*`alibi.utils.distance.squared_pairwise_distance(x, y, a_min=1e-07, a_max=1e+30)`

numpy pairwise squared Euclidean distance between samples *x* and *y*.

Parameters

- **x** (`ndarray`) – A batch of instances of shape *Nx x features*.
- **y** (`ndarray`) – A batch of instances of shape *Ny x features*.
- **a_min** (`float`) – Lower bound to clip distance values.
- **a_max** (`float`) – Upper bound to clip distance values.

Return type`ndarray`**Returns***Pairwise squared Euclidean distance *Nx x Ny*.*

alibi.utils.distributed module

class `alibi.utils.distributed.ActorPool(actors)`

Bases: `object`

__init__(*actors*)

Taken fom the *ray* repository: <https://github.com/ray-project/ray/pull/5945> . Create an actor pool from a list of existing actors. An actor pool is a utility class similar to *multiprocessing.Pool* that lets you schedule *ray* tasks over a fixed pool of actors.

Parameters

actors – List of *ray* actor handles to use in this pool.

Examples

```
>>> a1, a2 = Actor.remote(), Actor.remote()
>>> pool = ActorPool([a1, a2])
>>> print(pool.map(lambda a, v: a.double.remote(v), [1, 2, 3, 4]))
[2, 4, 6, 8]
```

get_next(*timeout=None*)

Returns the next pending result in order. This returns the next result produced by `alibi.utils.distributed.ActorPool.submit()`, blocking for up to the specified timeout until it is available.

Returns

The next result.

Raises

TimeoutError – If the timeout is reached.

Examples

```
>>> pool = ActorPool(...)
>>> pool.submit(lambda a, v: a.double.remote(v), 1)
>>> print(pool.get_next())
2
```

get_next_unordered(*timeout=None*)

Returns any of the next pending results. This returns some result produced by `alibi.utils.distributed.ActorPool.submit()`, blocking for up to the specified timeout until it is available. Unlike `alibi.utils.distributed.ActorPool.get_next()`, the results are not always returned in same order as submitted, which can improve performance.

Returns

The next result.

Raises

TimeoutError –

Examples

```
>>> pool = ActorPool(...)
>>> pool.submit(lambda a, v: a.double.remote(v), 1)
>>> pool.submit(lambda a, v: a.double.remote(v), 2)
>>> print(pool.get_next_unordered())
4
>>> print(pool.get_next_unordered())
2
```

has_next()

Returns whether there are any pending results to return.

Returns

True if there are any pending results not yet returned.

Examples

```
>>> pool = ActorPool(...)
>>> pool.submit(lambda a, v: a.double.remote(v), 1)
>>> print(pool.has_next())
True
>>> print(pool.get_next())
2
>>> print(pool.has_next())
False
```

map(fn, values, chunksize=1)

Apply the given function in parallel over the *actors* and *values*. This returns an ordered iterator that will return results of the map as they finish. Note that you must iterate over the iterator to force the computation to finish.

Parameters

- **fn** (*Callable*) – Function that takes (*actor*, *value*) as argument and returns an *ObjectID* computing the result over the *value*. The *actor* will be considered busy until the *ObjectID* completes.
- **values** (*list*) – List of values that *fn(actor, value)* should be applied to.
- **chunksize** (*int*) – Splits the list of values to be submitted to the parallel process into sublists of size *chunksize* or less.

Returns

Iterator over results from applying *fn* to the *actors* and *values*.

Examples

```
>>> pool = ActorPool(...)
>>> print(pool.map(lambda a, v: a.double.remote(v), [1, 2, 3, 4]))
[2, 4, 6, 8]
```

`map_unordered(fn, values, chunksize=1)`

Similar to `alibi.utils.distributed.ActorPool.map()`, but returning an unordered iterator. This returns an unordered iterator that will return results of the map as they finish. This can be more efficient than `alibi.utils.distributed.ActorPool.map()` if some results take longer to compute than others.

Parameters

- **fn** (*Callable*) – Function that takes (*actor*, *value*) as argument and returns an *ObjectID* computing the result over the *value*. The *actor* will be considered busy until the *ObjectID* completes.
- **values** (*list*) – List of values that *fn(actor, value)* should be applied to.
- **chunksize** (*int*) – Splits the list of values to be submitted to the parallel process into sublists of size *chunksize* or less.

Returns

Iterator over results from applying *fn* to the *actors* and *values*.

Examples

```
>>> pool = ActorPool(...)
>>> print(pool.map(lambda a, v: a.double.remote(v), [1, 2, 3, 4]))
[6, 2, 4, 8]
```

`submit(fn, value)`

Schedule a single task to run in the pool. This has the same argument semantics as `alibi.utils.distributed.ActorPool.map()`, but takes on a single value instead of a list of values. The result can be retrieved using `alibi.utils.distributed.ActorPool.get_next()` / `alibi.utils.distributed.ActorPool.get_next_unordered()`.

Parameters

- **fn** (*Callable*) – Function that takes (*actor*, *value*) as argument and returns an *ObjectID* computing the result over the *value*. The *actor* will be considered busy until the *ObjectID* completes.
- **value** (*object*) – Value to compute a result for.

Examples

```
>>> pool = ActorPool(...)
>>> pool.submit(lambda a, v: a.double.remote(v), 1)
>>> pool.submit(lambda a, v: a.double.remote(v), 2)
>>> print(pool.get_next(), pool.get_next())
2, 4
```

```
class alibi.utils.distributed.DistributedExplainer(distributed_opts, explainer_type,
                                                  explainer_init_args, explainer_init_kwargs,
                                                  concatenate_results=True,
                                                  return_generator=False)
```

Bases: `object`

A class that orchestrates the execution of the execution of a batch of explanations in parallel.

__getattr__(item)

Accesses actor attributes. Use sparingly as this involves a remote call (that is, these attributes are of an object in a different process). *The intended use is for retrieving any common state across the actor at the end of the computation in order to form the response (see notes 2 & 3).*

Parameters

item (`str`) – The explainer attribute to be returned.

Return type

`Any`

Returns

The value of the attribute specified by *item*.

Raises

ValueError – If the actor index is invalid.

Notes

1. This method assumes that the actor implements a *return_attribute* method.
2. Note that we are indexing the idle actors. This means that if a pool was initialised with 5 actors and 3 are busy, indexing with index 2 will raise an *IndexError*.
3. The order of *_idle_actors* constantly changes - an actor is removed from it if there is a task to execute and appended back when the task is complete. Therefore, indexing at the same position as computation proceeds will result in retrieving state from different processes.

```
__init__(distributed_opts, explainer_type, explainer_init_args, explainer_init_kwargs,
         concatenate_results=True, return_generator=False)
```

Creates a pool of actors (i.e., replicas of an instantiated *explainer_type* in a separate process) which can explain batches of instances in parallel via calls to *get_explanation*.

Parameters

- **distributed_opts** (`Dict[str, Any]`) – A dictionary with the following type (minimal signature):

```
class DistributedOpts(TypedDict):
    n_cpus: Optional[int]
    batch_size: Optional[int]
```

The dictionary may contain two additional keys:

- **'actor_cpu_frac'** : (`float`, `<= 1.0`, `>0.0`) - This is used to create more than one process on one CPU/GPU. This may not speed up CPU intensive tasks but it is worth experimenting with when few physical cores are available. In particular, this is highly useful when the user wants to share a GPU for multiple tasks, with the caveat that the machine learning framework itself needs to support running multiple replicas on the same GPU. See the [ray](#) documentation [here](#) for details.

- 'algorithm' : str - this is specified internally by the caller. It is used in order to register target function callbacks for the parallel pool. These should be implemented in the global scope. If not specified, its value will be 'default', which will select a default target function which expects the actor has a *get_explanation* method.

- **explainer_type** (Any) – Explainer class.
- **explainer_init_args** (Tuple) – Positional arguments to explainer constructor.
- **explainer_init_kwargs** (dict) – Keyword arguments to explainer constructor.
- **concatenate_results** (bool) – If True concatenates the results. See [alibi.utils.distributed.concatenate_minibatches\(\)](#) for more details.
- **return_generator** (bool) – If True a generator that returns the results in the order the computation finishes is returned when *get_explanation* is called. Otherwise, the order of the results is the same as the order of the minibatches.

Notes

When `return_generator=True`, the caller has to take elements from the generator (e.g., by calling *next*) in order to start computing the results (because the *ray* pool is implemented as a generator).

property actor_index: int

Returns the index of the actor for which state is returned.

Return type

int

concatenate: Callable

create_parallel_pool(explainer_type, explainer_init_args, explainer_init_kwargs)

Creates a pool of actors that can explain the rows of a dataset in parallel.

Parameters

documentation. (See *constructor*) –

get_explanation(X, **kwargs)

Performs distributed explanations of instances in X.

Parameters

- **X** (ndarray) – A batch of instances to be explained. Split into batches according to the settings passed to the constructor.
- ****kwargs** – Any keyword-arguments for the explainer *explain* method.

Return type

Union[Generator[Tuple[int, Any], None, None], List[Any], Any]

Returns

The explanations are returned as –

- a generator, if the *return_generator* option is specified. This is used so that the caller can access the results as they are computed. This is the only case when this method is non-blocking and the caller needs to call *next* on the generator to trigger the parallel computation.
- a list of objects, whose type depends on the return type of the explainer. This is returned if no custom preprocessing function is specified.

- an object, whose type depends on the return type of the concatenation function return when called with a list of minibatch results with the same order as the minibatches.

return_attribute(*name*)

Returns an attribute specified by its name. Used in a distributed context where the properties cannot be accessed using the dot syntax.

Return type

Any

set_actor_index(*value*)

Sets actor index. This is used when the *DistributedExplainer* is in a separate process because *ray* does not support calling property setters remotely

class `alibi.utils.distributed.PoolCollection`(*distributed_opts*, *explainer_type*, *explainer_init_args*, *explainer_init_kwargs*, ***kwargs*)

Bases: `object`

A wrapper object that turns a *DistributedExplainer* into a remote actor. This allows running multiple distributed explainers in parallel.

__getattr__(*item*)

Access attributes of the distributed explainer or the distributed explainer contained.

Return type

Any

__init__(*distributed_opts*, *explainer_type*, *explainer_init_args*, *explainer_init_kwargs*, ***kwargs*)

Initialises a list of *distinct* distributed explainers which can explain the same batch in parallel. It generalizes the *DistributedExplainer*, which contains replicas of one explainer object, speeding up the task of explaining batches of instances.

Parameters

- **distributed_opts** (`Dict[str, Any]`) – See `alibi.utils.distributed.DistributedExplainer()` constructor documentation for explanations. Each entry in the list is a different explainer configuration (e.g., CEM in PN vs PP mode, different background dataset sizes for SHAP, etc).
- **explainer_type** (`Any`) – See `alibi.utils.distributed.DistributedExplainer()` constructor documentation for explanations. Each entry in the list is a different explainer configuration (e.g., CEM in PN vs PP mode, different background dataset sizes for SHAP, etc).
- **explainer_init_args** (`List[Tuple]`) – See `alibi.utils.distributed.DistributedExplainer()` constructor documentation for explanations. Each entry in the list is a different explainer configuration (e.g., CEM in PN vs PP mode, different background dataset sizes for SHAP, etc).
- **explainer_init_kwargs** (`List[Dict]`) – See `alibi.utils.distributed.DistributedExplainer()` constructor documentation for explanations. Each entry in the list is a different explainer configuration (e.g., CEM in PN vs PP mode, different background dataset sizes for SHAP, etc).
- ****kwargs** – Any other kwargs, passed to the *DistributedExplainer* objects.

Raises

- **ResourceError** – If the number of CPUs specified by the user is smaller than the number of distributed explainers.

- **ValueError** – If the number of entries in the explainers args/kwargs list differ.

static `create_explainer_handles`(*distributed_opts*, *explainer_type*, *explainer_init_args*,
explainer_init_kwargs, ***kwargs*)

Creates multiple actors for *DistributedExplainer* so that tasks can be executed in parallel. The actors are initialised with different arguments, so they represent different explainers.

Parameters

- **distributed_opts** (`Dict[str, Any]`) – See `alibi.utils.distributed.PoolCollection()`.
- **explainer_type** (`Any`) – See `alibi.utils.distributed.PoolCollection()`.
- **explainer_init_args** (`List[Tuple]`) – See `alibi.utils.distributed.PoolCollection()`.
- **explainer_init_kwargs** (`List[Dict]`) – See `alibi.utils.distributed.PoolCollection()`.
- ****kwargs** – See `alibi.utils.distributed.PoolCollection()`.

get_explanation(*X*, ***kwargs*)

Calls a collection of distributed explainers in parallel. Each distributed explainer will explain each row in *X* in parallel.

Parameters

X – Batch of instances to be explained.

Return type

`List`

Returns

A list of responses collected from each explainer.

Notes

Note that the call to `ray.get` is blocking.

Raises

TypeError – If the user sets `return_generator=True` for the *DistributedExplainer*. This is because generators cannot be pickled so one cannot call `ray.get`.

property `remote_explainer_index`: `int`

Returns the index of the actor for which state is returned.

Return type

`int`

exception `alibi.utils.distributed.ResourceError`

Bases: `Exception`

`alibi.utils.distributed.batch`(*X*, *batch_size=None*, *n_batches=4*)

Splits the input into sub-arrays.

Parameters

- **X** (`ndarray`) – Array to be split.
- **batch_size** (`Optional[int]`) – The size of each batch. In particular

- if `batch_size` is not `None`, batches of this size are created. The sizes of the batches created might vary if the 0-th dimension of `X` is not divisible by `batch_size`. For an array of length `l` that should be split into `n` sections, it returns `l % n` sub-arrays of size `l//n + 1` and the rest of size `l//n`
- if `batch_size` is `None`, then `X` is split into `n_batches` sub-arrays.
- **n_batches** (`int`) – Number of batches in which to split the sub-array. Only used if `batch_size = None`

Return type`List[ndarray]`**Returns**A list of sub-arrays of `X`.`alibi.utils.distributed.concatenate_minibatches(minibatch_results)`

Merges the explanations computed on minibatches so that the distributed explainer returns the same output as the sequential version. If the type returned by the explainer is not supported by the function, expand this function by adding an appropriately named private function and use this function to check the input type and call it.

Parameters

minibatch_results (`Union[List[ndarray], List[List[ndarray]]`) – Explanations for each minibatch.

Return type`Union[ndarray, List[ndarray]]`**Returns**

- If the input is `List[np.ndarray]`, a single *numpy* array obtained by concatenating *mini-batch* results along the 0th axis.
- If the input is `List[List[np.ndarray]]` A list of *numpy* arrays obtained by concatenating arrays in with the same position in the sublists along the 0th axis.

`alibi.utils.distributed.default_target_fcn(actor, instances, kwargs=None)`

A target function that is executed in parallel given an actor pool. Its arguments must be an actor and a batch of values to be processed by the actor. Its role is to execute distributed computations when an actor is available.

Parameters

- **actor** (`Any`) – A *ray* actor. This is typically a class decorated with the `@ray.remote decorator`, that has been subsequently instantiated using `cls.remote(*args, **kwargs)`.
- **instances** (`tuple`) – A (`batch_index`, `batch`) tuple containing the batch of instances to be explained along with a batch index.
- **kwargs** (`Optional[Dict]`) – A list of keyword arguments for the actor `get_explanation` method.

Returns

A future that can be used to later retrieve the results of a distributed computation.

Notes

This function can be customized (e.g., if one does not desire to wrap the explainer such that it has *get_explanation* method. The customized function should be called **_target_fcn* with the wildcard being replaced by the name of the explanation method (e.g., *cem*, *cfproto*, etc). The same name should be added to the *distributed_opts* dictionary passed by the user prior to instantiating the *DistributedExplainer*.

`alibi.utils.distributed.invert_permutation(p)`

Inverts a permutation.

Parameters

p (`list`) – Some permutation of $0, 1, \dots, \text{len}(p)-1$. Returns an array *s*, where *s*[*i*] gives the index of *i* in *p*.

Return type

ndarray

Returns

s – *s*[*i*] gives the index of *i* in *p*.

`alibi.utils.distributed.order_result(unordered_result)`

Re-orders the result of a distributed explainer so that the explanations follow the same order as the input to the explainer.

Parameters

unordered_result (`Generator[Tuple[int, Any], None, None]`) – Each tuple contains the batch id as the first entry and the explanations for that batch as the second.

Return type

`List`

Returns

A list with re-ordered results.

Notes

This should not be used if one wants to take advantage of the results being returned as they are calculated.

alibi.utils.distributions module

`alibi.utils.distributions.kl_bernoulli(p, q)`

Compute KL-divergence between 2 probabilities *p* and *q*. *len(p)* divergences are calculated simultaneously.

Parameters

- **p** (ndarray) – Probability.
- **q** (ndarray) – Probability.

Return type

ndarray

Returns

Array with the KL-divergence between *p* and *q*.

alibi.utils.download module

`alibi.utils.download.spacy_model(model='en_core_web_md')`

Download *spaCy* model.

Parameters

model (`str`) – Model to be downloaded.

Return type

`None`

alibi.utils.frameworks module

`class alibi.utils.frameworks.Framework(value)`

Bases: `str`, `Enum`

An enumeration.

PYTORCH = `'pytorch'`

TENSORFLOW = `'tensorflow'`

alibi.utils.gradients module

`alibi.utils.gradients.num_grad_batch(func, X, args=(), eps=1e-08)`

Calculate the numerical gradients of a vector-valued function (typically a prediction function in classification) with respect to a batch of arrays *X*.

Parameters

- **func** (`Callable`) – Function to be differentiated.
- **X** (`ndarray`) – A batch of vectors at which to evaluate the gradient of the function.
- **args** (`Tuple`) – Any additional arguments to pass to the function.
- **eps** (`Union[float, ndarray]`) – Gradient step to use in the numerical calculation, can be a single *float* or one for each feature.

Return type

`ndarray`

Returns

An array of gradients at each point in the batch *X*.

`alibi.utils.gradients.perturb(X, eps=1e-08, proba=False)`

Apply perturbation to instance or prediction probabilities. Used for numerical calculation of gradients.

Parameters

- **X** (`ndarray`) – Array to be perturbed.
- **eps** (`Union[float, ndarray]`) – Size of perturbation.
- **proba** (`bool`) – If `True`, the net effect of the perturbation needs to be 0 to keep the sum of the probabilities equal to 1.

Return type

`Tuple[ndarray, ndarray]`

Returns

Instances where a positive and negative perturbation is applied.

alibi.utils.kernel module**class alibi.utils.kernel.EuclideanDistance**

Bases: `object`

`__call__(x, y)`

Computes the kernel distance matrix between x and y .

Parameters

- **x** (ndarray) – The first array of data instances.
- **y** (ndarray) – The second array of data instances.

Return type

ndarray

Returns

Kernel distance matrix between x and y having the size of $N_x \times N_y$, where N_x is the number of instances in x and y is the number of instances in y .

`__init__()`

Euclidean distance: $k(x, y) = ||x - y||$. A forward pass takes a batch of instances x of size $N_x \times f1 \times f2 \times \dots$ and y of size $N_y \times f1 \times f2 \times \dots$ and returns the kernel matrix $N_x \times N_y$.

class alibi.utils.kernel.GaussianRBF(sigma=None)

Bases: `object`

`__call__(x, y, infer_sigma=False)`

Computes the kernel matrix between x and y .

Parameters

- **x** (ndarray) – The first array of data instances.
- **y** (ndarray) – The second array of data instances.
- **infer_sigma** (bool) – Whether to infer *sigma* automatically. The *sigma* value is computed based on the median distance value between the instances from x and y .

Return type

ndarray

Returns

Kernel matrix between x and y having the size of $N_x \times N_y$ where N_x is the number of instances in x and y is the number of instances in y .

`__init__(sigma=None)`

Gaussian RBF kernel: $k(x, y) = \exp(-\frac{||x-y||^2}{2\sigma^2})$. A forward pass takes a batch of instances x of size $N_x \times f1 \times f2 \times \dots$ and y of size $N_y \times f1 \times f2 \times \dots$ and returns the kernel matrix of size $N_x \times N_y$.

Parameters

sigma (Union[float, ndarray, None]) – Kernel bandwidth. Not to be specified if being inferred or trained. Can pass multiple values to evaluate the kernel with and then average.

property **sigma**: ndarray

Return type
ndarray

class `alibi.utils.kernel.GaussianRBFDistance(sigma=None)`

Bases: `object`

__init__(sigma=None)

Gaussian RBF kernel dissimilarity/distance: $k(x, y) = 1 - \exp(-\frac{\|x-y\|^2}{2\sigma^2})$. A forward pass takes a batch of instances x of size $N_x \times f1 \times f2 \times \dots$ and y of size $N_y \times f1 \times f2 \times \dots$ and returns the kernel matrix of size $N_x \times N_y$.

Parameters

sigma (`Union[float, ndarray, None]`) – See `alibi.utils.kernel.GaussianRBF.__init__()`.

alibi.utils.lang_model module

This module defines a wrapper for transformer-based masked language models used in *AnchorText* as a perturbation strategy. The *LanguageModel* base class defines basic functionalities as loading, storing, and predicting.

Language model's tokenizers usually work at a subword level, and thus, a word can be split into subwords. For example, a word can be decomposed as: `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`. For language models such as *DistilbertBaseUncased* and *BertBaseUncased*, the tail tokens can be identified by a special prefix '##'. On the other hand, for *RobertaBase* only the head is prefixed with the special character 'Ġ', thus the tail tokens can be identified by the absence of the special token. In this module, we refer to a tail token as a subword prefix. We will use the notion of a subword to refer to either a *head* or a *tail* token.

To generate interpretable perturbed instances, we do not mask subwords, but entire words. Note that this operation is equivalent to replacing the head token with the special mask token, and removing the tail tokens if they exist. Thus, the *LanguageModel* class offers additional functionalities such as: checking if a token is a subword prefix, selection of a word (head_token along with the tail_tokens), etc.

Some language models can work with a limited number of tokens, thus the input text has to be split. Thus, a text will be split in head and tail, where the number of tokens in the head is less or equal to the maximum allowed number of tokens to be processed by the language model. In the *AnchorText* only the head is perturbed. To keep the results interpretable, we ensure that the head will not end with a subword, and will contain only full words.

class `alibi.utils.lang_model.BertBaseUncased(preloading=True)`

Bases: `LanguageModel`

SUBWORD_PREFIX = '##'

Language model subword prefix.

__init__(preloading=True)

Initialize *BertBaseUncased*.

Parameters

preloading (`bool`) – See `alibi.utils.lang_model.LanguageModel.__init__()`.

caller: `Callable`

is_subword_prefix(token)

Checks if the given token is a part of the tail of a word. Note that a word can be split in multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). Each language model has a convention on how to mark a tail token. For example *DistilbertBaseUncased* and *BertBaseUncased*

have the tail tokens prefixed with the special set of characters '##'. On the other hand, for *RobertaBase* only the head token is prefixed with the special character 'Ġ' and thus we need to check the absence of the prefix to identify the tail tokens. We call those special characters *SUBWORD_PREFIX*. Due to different conventions, this method has to be implemented for each language model. See module docstring for namings.

Parameters

token (`str`) – Token to be checked if it is a subword.

Return type

`bool`

Returns

True if the given token is a subword prefix. False otherwise.

property mask: `str`

Returns the mask token.

Return type

`str`

model: `Any`

tokenizer: `Any`

class `alibi.utils.lang_model.DistilbertBaseUncased(preloading=True)`

Bases: `LanguageModel`

SUBWORD_PREFIX = '##'

Language model subword prefix.

__init__(`preloading=True`)

Initialize *DistilbertBaseUncased*.

Parameters

preloading (`bool`) – See `alibi.utils.lang_model.LanguageModel.__init__()`.

caller: `Callable`

is_subword_prefix(`token`)

Checks if the given token is a part of the tail of a word. Note that a word can be split in multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). Each language model has a convention on how to mark a tail token. For example *DistilbertBaseUncased* and *BertBaseUncased* have the tail tokens prefixed with the special set of characters '##'. On the other hand, for *RobertaBase* only the head token is prefixed with the special character 'Ġ' and thus we need to check the absence of the prefix to identify the tail tokens. We call those special characters *SUBWORD_PREFIX*. Due to different conventions, this method has to be implemented for each language model. See module docstring for namings.

Parameters

token (`str`) – Token to be checked if it is a subword.

Return type

`bool`

Returns

True if the given token is a subword prefix. False otherwise.

property mask: `str`

Returns the mask token.

Return type

`str`

model: `Any`

tokenizer: `Any`

class `alibi.utils.lang_model.LanguageModel(model_path, preloading=True)`

Bases: `ABC`

SUBWORD_PREFIX = ''

Language model subword prefix.

__init__(`model_path`, `preloading=True`)

Initialize the language model.

Parameters

- **model_path** (`str`) – *transformers* package model path.
- **preloading** (`bool`) – Whether to preload the online version of the transformer. If False, a call to *from_disk* method is expected.

caller: `Callable`

from_disk(`path`)

Loads a model from disk.

Parameters

path (`Union[str, Path]`) – Path to the checkpoint.

head_tail_split(`text`)

Split the text in head and tail. Some language models support a maximum number of tokens. Thus is necessary to split the text to meet this constraint. After the text is split in head and tail, only the head is considered for operation. Thus the tail will remain unchanged.

Parameters

text (`str`) – Text to be split in head and tail.

Return type

`Tuple[str, str, List[str], List[str]]`

Returns

Tuple consisting of the head, tail and their corresponding list of tokens.

is_punctuation(`token`, `punctuation`)

Checks if the given token is punctuation.

Parameters

- **token** (`str`) – Token to be checked if it is punctuation.
- **punctuation** (`str`) – String containing all punctuation to be considered.

Return type

`bool`

Returns

True if the *token* is a punctuation. False otherwise.

is_stop_word(*tokenized_text*, *start_idx*, *punctuation*, *stopwords*)

Checks if the given word starting at the given index is in the list of stopwords.

Parameters

- **tokenized_text** (`List[str]`) – Tokenized text.
- **start_idx** (`int`) – Starting index of a word.
- **stopwords** (`Optional[List[str]]`) – List of stop words. The words in this list should be lowercase.
- **punctuation** (`str`) – Punctuation to be considered. See `alibi.utils.lang_model.LanguageModel.select_entire_word()`.

Return type

`bool`

Returns

True if the *token* is in the *stopwords* list. False otherwise.

abstract is_subword_prefix(*token*)

Checks if the given token is a part of the tail of a word. Note that a word can be split in multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). Each language model has a convention on how to mark a tail token. For example *DistilbertBaseUncased* and *BertBaseUncased* have the tail tokens prefixed with the special set of characters '##'. On the other hand, for *RobertaBase* only the head token is prefixed with the special character 'Ġ' and thus we need to check the absence of the prefix to identify the tail tokens. We call those special characters *SUBWORD_PREFIX*. Due to different conventions, this method has to be implemented for each language model. See module docstring for namings.

Parameters

token (`str`) – Token to be checked if it is a subword.

Return type

`bool`

Returns

True if the given token is a subword prefix. False otherwise.

abstract property mask: `str`

Returns the mask token.

Return type

`str`

property mask_id: `int`

Returns the mask token id

Return type

`int`

property max_num_tokens: `int`

Returns the maximum number of token allowed by the model.

Return type

`int`

model: `Any`

predict_batch_lm(*x*, *vocab_size*, *batch_size*)

Tensorflow language model batch predictions for *AnchorText*.

Parameters

- **x** (*BatchEncoding*) – Batch of instances.
- **vocab_size** (*int*) – Vocabulary size of language model.
- **batch_size** (*int*) – Batch size used for predictions.

Return type

ndarray

Returns

y – Array with model predictions.

select_word(*tokenized_text*, *start_idx*, *punctuation*)

Given a tokenized text and the starting index of a word, the function selects the entire word. Note that a word is composed of multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). The tail tokens can be identified based on the presence/absence of *SUBWORD_PREFIX*. See `alibi.utils.lang_model.LanguageModel.is_subword_prefix()` for more details.

Parameters

- **tokenized_text** (*List[str]*) – Tokenized text.
- **start_idx** (*int*) – Starting index of a word.
- **punctuation** (*str*) – String of punctuation to be considered. If it encounters a token composed only of characters in *punctuation* it terminates the search.

Return type

str

Returns

The word obtained by concatenation `[head_token tail_token_1 tail_token_2 ... tail_token_k]`.

to_disk(*path*)

Saves a model to disk.

Parameters

path (*Union[str, Path]*) – Path to the checkpoint.

tokenizer: *Any*

class `alibi.utils.lang_model.RobertaBase`(*preloading=True*)

Bases: *LanguageModel*

SUBWORD_PREFIX = 'Ġ'

Language model subword prefix.

__init__(*preloading=True*)

Initialize *RobertaBase*.

Parameters

preloading (*bool*) – See `alibi.utils.lang_model.LanguageModel.__init__()` constructor.

caller: *Callable*

is_subword_prefix(*token*)

Checks if the given token is a part of the tail of a word. Note that a word can be split in multiple tokens (e.g., `word = [head_token tail_token_1 tail_token_2 ... tail_token_k]`). Each language model has a convention on how to mark a tail token. For example *DistilbertBaseUncased* and *BertBaseUncased* have the tail tokens prefixed with the special set of characters '##'. On the other hand, for *RobertaBase* only the head token is prefixed with the special character 'Ġ' and thus we need to check the absence of the prefix to identify the tail tokens. We call those special characters *SUBWORD_PREFIX*. Due to different conventions, this method has to be implemented for each language model. See module docstring for namings.

Parameters

token (`str`) – Token to be checked if it is a subword.

Return type

`bool`

Returns

True if the given token is a subword prefix. False otherwise.

property mask: `str`

Returns the mask token.

Return type

`str`

model: `Any`

tokenizer: `Any`

alibi.utils.mapping module**alibi.utils.mapping.num_to_ord**(*data, dist*)

Transform numerical values into categories using the map calculated under the fit method.

Parameters

- **data** (`ndarray`) – *Numpy* array with the numerical data.
- **dist** (`dict`) – Dict with as keys the categorical variables and as values the numerical value for each category.

Return type

`ndarray`

Returns

Numpy array with transformed numerical data into categories.

alibi.utils.mapping.ohe_to_ord(*X_ohe, cat_vars_ohe*)

Convert one-hot encoded variables to ordinal encodings.

Parameters

- **X_ohe** (`ndarray`) – Data with mixture of one-hot encoded and numerical variables.
- **cat_vars_ohe** (`dict`) – Dict with as keys the first column index for each one-hot encoded categorical variable and as values the number of categories per categorical variable.

Return type

`Tuple[ndarray, dict]`

Returns

Ordinal equivalent of one-hot encoded data and dict with categorical columns and number of categories.

`alibi.utils.mapping.ohc_to_ord_shape(shape, cat_vars, is_ohc=False)`

Infer shape of instance if the categorical variables have ordinal instead of one-hot encoding.

Parameters

- **shape** (`tuple`) – Instance shape, starting with batch dimension.
- **cat_vars** (`Dict[int, int]`) – Dict with as keys the categorical columns and as values the number of categories per categorical variable.
- **is_ohc** (`bool`) – Whether instance is OHE.

Return type

`tuple`

Returns

Tuple with shape of instance with ordinal encoding of categorical variables.

`alibi.utils.mapping.ord_to_num(data, dist)`

Transform categorical into numerical values using a mapping.

Parameters

- **data** (`ndarray`) – *Numpy* array with the categorical data.
- **dist** (`dict`) – Dict with as keys the categorical variables and as values the numerical value for each category.

Return type

`ndarray`

Returns

Numpy array with transformed categorical data into numerical values.

`alibi.utils.mapping.ord_to_ohc(X_ord, cat_vars_ord)`

Convert ordinal to one-hot encoded variables.

Parameters

- **X_ord** (`ndarray`) – Data with mixture of ordinal encoded and numerical variables.
- **cat_vars_ord** (`dict`) – Dict with as keys the categorical columns and as values the number of categories per categorical variable.

Return type

`Tuple[ndarray, dict]`

Returns

One-hot equivalent of ordinal encoded data and dict with categorical columns and number of categories.

alibi.utils.missing_optional_dependency module

Functionality for optional importing

This module provides a way to import optional dependencies. In the case that the user imports some functionality from alibi that is not usable due to missing optional dependencies this code is used to allow the import but replace it with an object that throws an error on use. This way we avoid errors at import time that prevent the user using functionality independent of the missing dependency.

```
class alibi.utils.missing_optional_dependency.MissingDependency(object_name, err,
                                                             missing_dependency='all')
```

Bases: `object`

Missing Dependency Class

Used to replace any object that requires unmet optional dependencies. Attribute access or calling the `__call__` method on this object will raise an error.

```
__call__(*args, **kwargs)
```

If called, raise an error.

```
__getattr__(key)
```

Raise an error when attributes are accessed.

```
__init__(object_name, err, missing_dependency='all')
```

Metaclass for MissingDependency classes

Parameters

- **object_name** (`str`) – Name of object we are replacing
- **missing_dependency** (`str`) – Name of missing dependency required for object
- **err** (`Union[ModuleNotFoundError, ImportError]`) – Error to be raised when the class is initialized or used

```
property err_msg
```

Generate error message informing user to install missing dependencies.

```
alibi.utils.missing_optional_dependency.err_msg_template = <string.Template object>
```

Mapping used to ensure correct pip install message is generated if a missing optional dependency is detected. This dict is used to control two behaviours:

1. When we import objects from missing dependencies we check that any *ModuleNotFoundError* or *ImportError* corresponds to a missing optional dependency by checking the name of the missing dependency is in *ERROR_TYPES*. We then map this name to the corresponding optional dependency bucket that will resolve the issue.
2. Some optional dependencies have multiple names such as *torch* and *pytorch*, instead of enforcing a single naming convention across the whole code base we instead use *ERROR_TYPES* to capture both cases. This is done right before the pip install message is issued as this is the most robust place to capture these differences.

```
alibi.utils.missing_optional_dependency.import_optional(module_name, names=None)
```

Import a module that depends on optional dependencies

Note: This function is used to import modules that depend on optional dependencies. Because it mirrors the python import functionality its return type has to be *Any*. Using objects imported with this function can lead to misspecification of types as *Any* when the developer intended to be more restrictive.

Parameters

- **module_name** (`str`) – The module to import
- **names** (`Optional[List[str]]`) – The names to import from the module. If `None`, all names are imported.

Return type`Any`**Returns**

- *The module or named objects within the modules if names is not None. If the import fails due to a*
- *ModuleNotFoundError or ImportError then the requested module or named objects are replaced with instances of*
- *the MissingDependency class above.*

alibi.utils.tf module`alibi.utils.tf.argmax_grad(x)``alibi.utils.tf.argmin_grad(x, y)``alibi.utils.tf.one_hot_grad(x, y)``alibi.utils.tf.round_grad(x)`**alibi.utils.visualization module**`class alibi.utils.visualization.ImageVisualizationMethod(value)`Bases: `Enum`

An enumeration.

`alpha_scaling = 5``blended_heat_map = 2``heat_map = 1``masked_image = 4``original_image = 3``class alibi.utils.visualization.VisualizeSign(value)`Bases: `Enum`

An enumeration.

`absolute_value = 2``all = 4``negative = 3``positive = 1`

```
alibi.utils.visualization.heatmap(data, xticklabels, yticklabels, vmin=None, vmax=None, cmap='magma',
                                   robust=False, annot=True, linewidths=3, linecolor='w', cbar=True,
                                   cbar_label="", cbar_ax=None, cbar_kws=None, fmt='{x:.2f}',
                                   textcolors=('white', 'black'), threshold=None, text_kws=None, ax=None,
                                   **kwargs)
```

Constructs a heatmap with annotation.

Parameters

- **data** (ndarray) – A 2D *numpy* array of shape $M \times N$.
- **yticklabels** (List[str]) – A list or array of length M with the labels for the rows.
- **xticklabels** (List[str]) – A list or array of length N with the labels for the columns.
- **vmin** (Optional[float]) – When using scalar data and no explicit norm, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin/vmax* when norm is given. When using RGB(A) data, parameters *vmin/vmax* are ignored.
- **vmax** (Optional[float]) – When using scalar data and no explicit norm, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin/vmax* when norm is given. When using RGB(A) data, parameters *vmin/vmax* are ignored.
- **cmap** (Union[str, Colormap]) – The Colormap instance or registered colormap name used to map scalar data to colors. This parameter is ignored for RGB(A) data.
- **robust** (Optional[bool]) – If True and *vmin* or *vmax* are absent, the colormap range is computed with robust quantiles instead of the extreme values. Uses `numpy.nanpercentile` with *q* values set to 2 and 98, respectively.
- **annot** (Optional[bool]) – Boolean flag whether to annotate the heatmap. Default True.
- **linewidths** (float) – Width of the lines that will divide each cell. Default 3.
- **linecolor** (str) – Color of the lines that will divide each cell. Default "w".
- **cbar** (bool) – Boolean flag whether to draw a colorbar.
- **cbar_label** (str) – Optional label for the colorbar.
- **cbar_ax** (Optional[Axes]) – Optional axes in which to draw the colorbar, otherwise take space from the main axes.
- **cbar_kws** (Optional[dict]) – An optional dictionary with arguments to `matplotlib.figure.Figure.colorbar`.
- **fmt** (Union[str, Formatter]) – Format of the annotations inside the heatmap. This should either use the string format method, e.g. "{x:.2f}", or be a `matplotlib.ticker.Formatter`. Default "{x:.2f}".
- **textcolors** (Tuple[str, str]) – A tuple of *matplotlib* colors. The first is used for values below a threshold, the second for those above. Default ("black", "white").
- **threshold** (Optional[float]) – Optional value in data units according to which the colors from textcolors are applied. If None (the default) uses the middle of the colormap as separation.
- **text_kws** (Optional[dict]) – An optional dictionary with arguments to `matplotlib.axes.Axes.text`.
- **ax** (Optional[Axes]) – Axes in which to draw the plot, otherwise use the currently-active axes.

- **kwargs** – All other keyword arguments are passed to `matplotlib.axes.Axes.imshow`.

Return type

Axes

Returns*Axes object with the heatmap.*

```
alibi.utils.visualization.visualize_image_attr(attr, original_image=None, method='heat_map',
                                              sign='absolute_value', plt_fig_axis=None,
                                              outlier_perc=2, cmap=None, alpha_overlay=0.5,
                                              show_colorbar=False, title=None, fig_size=(6, 6),
                                              use_pyplot=True)
```

Visualizes attribution for a given image by normalizing attribution values of the desired sign ('positive' | 'negative' | 'absolute_value' | 'all') and displaying them using the desired mode in a *matplotlib* figure.

Parameters

- **attr** (ndarray) – *Numpy* array corresponding to attributions to be visualized. Shape must be in the form (H, W, C) , with channels as last dimension. Shape must also match that of the original image if provided.
- **original_image** (Optional[ndarray]) – *Numpy* array corresponding to original image. Shape must be in the form (H, W, C) , with channels as the last dimension. Image can be provided either with *float* values in range 0-1 or *int* values between 0-255. This is a necessary argument for any visualization method which utilizes the original image.
- **method** (str) – Chosen method for visualizing attribution. Supported options are:
 - 'heat_map' - Display heat map of chosen attributions
 - 'blended_heat_map' - Overlay heat map over greyscale version of original image. Parameter `alpha_overlay` corresponds to alpha of heat map.
 - 'original_image' - Only display original image.
 - 'masked_image' - Mask image (pixel-wise multiply) by normalized attribution values.
 - 'alpha_scaling' - Sets alpha channel of each pixel to be equal to normalized attribution value.
 Default: 'heat_map'.
- **sign** (str) – Chosen sign of attributions to visualize. Supported options are:
 - 'positive' - Displays only positive pixel attributions.
 - 'absolute_value' - Displays absolute value of attributions.
 - 'negative' - Displays only negative pixel attributions.
 - 'all' - Displays both positive and negative attribution values. This is not supported for 'masked_image' or 'alpha_scaling' modes, since signed information cannot be represented in these modes.
- **plt_fig_axis** (Optional[Tuple[Figure, Axes]]) – Tuple of *matplotlib.pyplot.figure* and *axis* on which to visualize. If *None* is provided, then a new figure and axis are created.
- **outlier_perc** (Union[int, float]) – Top attribution values which correspond to a total of *outlier_perc* percentage of the total attribution are set to 1 and scaling is performed using the minimum of these values. For `sign='all'`, outliers and scale value are computed using absolute value of attributions.

- **cmap** (*Optional*[*str*]) – String corresponding to desired colormap for heatmap visualization. This defaults to 'Reds' for negative sign, 'Blues' for absolute value, 'Greens' for positive sign, and a spectrum from red to green for all. Note that this argument is only used for visualizations displaying heatmaps.
- **alpha_overlay** (*float*) – Visualizes attribution for a given image by normalizing attribution values of the desired sign (positive, negative, absolute value, or all) and displaying them using the desired mode in a matplotlib figure.
- **show_colorbar** (*bool*) – Displays colorbar for heatmap below the visualization. If given method does not use a heatmap, then a colormap axis is created and hidden. This is necessary for appropriate alignment when visualizing multiple plots, some with colorbars and some without.
- **title** (*Optional*[*str*]) – The title for the plot. If *None*, no title is set.
- **fig_size** (*Tuple*[*int*, *int*]) – Size of figure created.
- **use_pyplot** (*bool*) – If *True*, uses *pyplot* to create and show figure and displays the figure after creating. If *False*, uses *matplotlib* object-oriented API and simply returns a figure object without showing.

Return type*Tuple*[*Figure*, *Axes*]**Returns***2-element tuple of consisting of –*

- *figure* : *matplotlib.pyplot.Figure* - Figure object on which visualization is created. If *plt_fig_axis* argument is given, this is the same figure provided.
- *axis* : *matplotlib.pyplot.Axes* - Axes object on which visualization is created. If *plt_fig_axis* argument is given, this is the same axis provided.

alibi.utils.wrappers module**class** *alibi.utils.wrappers.ArgmaxTransformer*(*predictor*)Bases: *object*

A transformer for converting classification output probability tensors to class labels. It assumes the predictor is a callable that can be called with a *N*-tensor of data points *x* and produces an *N*-tensor of outputs.

class *alibi.utils.wrappers.Predictor*(*clf*, *preprocessor=None*)Bases: *object**alibi.utils.wrappers.methoddispatch*(*func*)

A decorator that is used to support singledispatch style functionality for instance methods. By default, singledispatch selects a function to call from registered based on the type of *args[0]*:

```
def wrapper(*args, **kw):
    return dispatch(args[0].__class__)(*args, **kw)
```

This uses singledispatch to do achieve this but instead uses *args[1]* since *args[0]* will always be self.

13.1.2 Submodules

alibi.exceptions module

This module defines the Alibi exception hierarchy and common exceptions used across the library.

exception `alibi.exceptions.AlibiException(message)`

Bases: `Exception`, `ABC`

Abstract base class of all alibi exceptions.

class `alibi.exceptions.AlibiPredictorCallException`

Bases: `object`

class `alibi.exceptions.AlibiPredictorReturnTypeError`

Bases: `object`

exception `alibi.exceptions.NotFittedError(object_name)`

Bases: `AlibiException`

This exception is raised whenever a compulsory call to a *fit* method has not been carried out.

exception `alibi.exceptions.PredictorCallError(message)`

Bases: `AlibiException`, `AlibiPredictorCallException`

This exception is raised whenever a call to a user supplied predictor fails at runtime.

exception `alibi.exceptions.PredictorReturnTypeError(message)`

Bases: `AlibiException`, `AlibiPredictorReturnTypeError`

This exception is raised whenever the return type of a user supplied predictor is of an unexpected or unsupported type.

exception `alibi.exceptions.SerializationError(message)`

Bases: `AlibiException`

This exception is raised whenever an explainer cannot be serialized.

alibi.saving module

class `alibi.saving.NumpyEncoder(*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None)`

Bases: `JSONEncoder`

default (*obj*)

Implement this method in a subclass such that it returns a serializable object for *o*, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
```

(continues on next page)

(continued from previous page)

```
    return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

`alibi.saving.load_explainer(path, predictor)`

Load an explainer from disk.

Parameters

- **path** (`Union[str, PathLike]`) – Path to a directory containing the saved explainer.
- **predictor** – Model or prediction function used to originally initialize the explainer.

Return type

Explainer

Returns

An explainer instance.

`alibi.saving.save_explainer(explainer, path)`

Save an explainer to disk. Uses the *dill* module.

Parameters

- **explainer** (*Explainer*) – Explainer instance to save to disk.
- **path** (`Union[str, PathLike]`) – Path to a directory. A new directory will be created if one does not exist.

Return type

`None`

alibi.version module

PYTHON MODULE INDEX

a

- alibi, 561
- alibi.api, 561
- alibi.api.defaults, 561
- alibi.api.interfaces, 564
- alibi.confidence, 566
- alibi.confidence.model_linearity, 569
- alibi.confidence.trustscore, 570
- alibi.datasets, 572
- alibi.datasets.default, 574
- alibi.datasets.tensorflow, 576
- alibi.exceptions, 779
- alibi.explainers, 576
- alibi.explainers.ale, 678
- alibi.explainers.anchors, 619
- alibi.explainers.anchors.anchor_base, 619
- alibi.explainers.anchors.anchor_explanation, 624
- alibi.explainers.anchors.anchor_image, 625
- alibi.explainers.anchors.anchor_tabular, 630
- alibi.explainers.anchors.anchor_tabular_distributed, 637
- alibi.explainers.anchors.anchor_text, 640
- alibi.explainers.anchors.language_model_text_sampler, 644
- alibi.explainers.anchors.text_samplers, 647
- alibi.explainers.backends, 650
- alibi.explainers.backends.cfml_base, 664
- alibi.explainers.backends.cfml_tabular, 665
- alibi.explainers.backends.pytorch, 650
- alibi.explainers.backends.pytorch.cfml_base, 650
- alibi.explainers.backends.pytorch.cfml_tabular, 655
- alibi.explainers.backends.tensorflow, 657
- alibi.explainers.backends.tensorflow.cfml_base, 657
- alibi.explainers.backends.tensorflow.cfml_tabular, 663
- alibi.explainers.cem, 682
- alibi.explainers.cfproto, 685
- alibi.explainers.cfml_base, 689
- alibi.explainers.cfml_tabular, 695
- alibi.explainers.counterfactual, 698
- alibi.explainers.integrated_gradients, 700
- alibi.explainers.partial_dependence, 702
- alibi.explainers.pd_variance, 708
- alibi.explainers.permutation_importance, 711
- alibi.explainers.shap_wrappers, 715
- alibi.explainers.similarity, 673
- alibi.explainers.similarity.backends, 673
- alibi.explainers.similarity.backends.pytorch, 673
- alibi.explainers.similarity.backends.pytorch.base, 673
- alibi.explainers.similarity.backends.tensorflow, 673
- alibi.explainers.similarity.backends.tensorflow.base, 673
- alibi.explainers.similarity.base, 674
- alibi.explainers.similarity.grad, 675
- alibi.explainers.similarity.metrics, 677
- alibi.models, 722
- alibi.models.pytorch, 722
- alibi.models.pytorch.actor_critic, 722
- alibi.models.pytorch.autoencoder, 723
- alibi.models.pytorch.cfml_models, 724
- alibi.models.pytorch.metrics, 727
- alibi.models.pytorch.model, 728
- alibi.models.tensorflow, 730
- alibi.models.tensorflow.actor_critic, 730
- alibi.models.tensorflow.autoencoder, 731
- alibi.models.tensorflow.cfml_models, 733
- alibi.prototypes, 735
- alibi.prototypes.protoselect, 737
- alibi.saving, 779
- alibi.tests, 741
- alibi.tests.utils, 741
- alibi.utils, 742
- alibi.utils.approximation_methods, 751
- alibi.utils.data, 752
- alibi.utils.discretizer, 752
- alibi.utils.distance, 753
- alibi.utils.distributed, 756

- alibi.utils.distributions, [764](#)
- alibi.utils.download, [765](#)
- alibi.utils.frameworks, [765](#)
- alibi.utils.gradients, [765](#)
- alibi.utils.kernel, [766](#)
- alibi.utils.lang_model, [767](#)
- alibi.utils.mapping, [772](#)
- alibi.utils.missing_optional_dependency, [774](#)
- alibi.utils.tf, [775](#)
- alibi.utils.visualization, [775](#)
- alibi.utils.wrappers, [778](#)
- alibi.version, [780](#)

Symbols

<code>__attrs_post_init__()</code>	(alibi.api.interfaces.Explanation method), 564	<code>__init__()</code>	(alibi.confidence.LinearityMeasure method), 566
<code>__call__()</code>	(alibi.explainers.anchors.anchor_image.AnchorImageSampler method), 628	<code>__init__()</code>	(alibi.confidence.TrustScore method), 566
<code>__call__()</code>	(alibi.explainers.anchors.anchor_tabular.TabularSampler method), 633	<code>__init__()</code>	(alibi.confidence.model_linearity.LinearityMeasure method), 569
<code>__call__()</code>	(alibi.explainers.anchors.anchor_tabular_distributed.RemoteSampler method), 639	<code>__init__()</code>	(alibi.confidence.trustscore.TrustScore method), 570
<code>__call__()</code>	(alibi.explainers.anchors.language_model_text_sampler.LanguageModelTextSampler method), 644	<code>__init__()</code>	(alibi.explainers.ALE method), 576
<code>__call__()</code>	(alibi.explainers.anchors.text_samplers.SimilaritySampler method), 648	<code>__init__()</code>	(alibi.explainers.AnchorImage method), 578
<code>__call__()</code>	(alibi.explainers.anchors.text_samplers.UnknownSampler method), 649	<code>__init__()</code>	(alibi.explainers.AnchorTabular method), 581
<code>__call__()</code>	(alibi.explainers.cfml_base.Callback method), 689	<code>__init__()</code>	(alibi.explainers.AnchorText method), 584
<code>__call__()</code>	(alibi.explainers.cfml_base.NormalActionNoise method), 693	<code>__init__()</code>	(alibi.explainers.CEM method), 587
<code>__call__()</code>	(alibi.explainers.cfml_base.Postprocessing method), 693	<code>__init__()</code>	(alibi.explainers.Counterfactual method), 590
<code>__call__()</code>	(alibi.explainers.cfml_base.ConcatTabularPostprocessing method), 695	<code>__init__()</code>	(alibi.explainers.CounterfactualProto method), 591
<code>__call__()</code>	(alibi.explainers.cfml_base.SampleTabularPostprocessing method), 697	<code>__init__()</code>	(alibi.explainers.CounterfactualRL method), 595
<code>__call__()</code>	(alibi.models.pytorch.metrics.LossContainer method), 727	<code>__init__()</code>	(alibi.explainers.CounterfactualRLTabular method), 597
<code>__call__()</code>	(alibi.utils.kernel.EuclideanDistance method), 766	<code>__init__()</code>	(alibi.explainers.GradientSimilarity method), 600
<code>__call__()</code>	(alibi.utils.kernel.GaussianRBF method), 766	<code>__init__()</code>	(alibi.explainers.IntegratedGradients method), 602
<code>__call__()</code>	(alibi.utils.missing_optional_dependency.MissingDependency method), 774	<code>__init__()</code>	(alibi.explainers.KernelShap method), 604
<code>__getattr__()</code>	(alibi.utils.DistributedExplainer method), 743	<code>__init__()</code>	(alibi.explainers.PartialDependence method), 606
<code>__getattr__()</code>	(alibi.utils.distributed.DistributedExplainer method), 759	<code>__init__()</code>	(alibi.explainers.PartialDependenceVariance method), 608
<code>__getattr__()</code>	(alibi.utils.distributed.PoolCollection method), 761	<code>__init__()</code>	(alibi.explainers.PermutationImportance method), 610
<code>__getattr__()</code>	(alibi.utils.missing_optional_dependency.MissingDependency method), 774	<code>__init__()</code>	(alibi.explainers.TreePartialDependence method), 612
<code>__getitem__()</code>	(alibi.api.interfaces.Explanation method), 624	<code>__init__()</code>	(alibi.explainers.TreeShap method), 613
		<code>__init__()</code>	(alibi.explainers.ale.ALE method), 678
		<code>__init__()</code>	(alibi.explainers.anchors.anchor_base.AnchorBaseBeam method), 619
		<code>__init__()</code>	(alibi.explainers.anchors.anchor_explanation.AnchorExplanation method), 624

`__init__()` (`alibi.explainers.anchors.anchor_image.AnchorImage` method), 718
`method`), 625 `__init__()` (`alibi.explainers.similarity.base.BaseSimilarityExplainer`
`__init__()` (`alibi.explainers.anchors.anchor_image.AnchorImageSampler` method), 674
`method`), 628 `__init__()` (`alibi.explainers.similarity.grad.GradientSimilarity`
`__init__()` (`alibi.explainers.anchors.anchor_tabular.AnchorTabular` method), 675
`method`), 630 `__init__()` (`alibi.models.pytorch.actor_critic.Actor`
`__init__()` (`alibi.explainers.anchors.anchor_tabular.TabularSampler` method), 722
`method`), 633 `__init__()` (`alibi.models.pytorch.actor_critic.Critic`
`__init__()` (`alibi.explainers.anchors.anchor_text.AnchorText` method), 723
`method`), 640 `__init__()` (`alibi.models.pytorch.autoencoder.AE`
`__init__()` (`alibi.explainers.anchors.language_model_text_sampler.LanguageModelSampler`
`method`), 645 `__init__()` (`alibi.models.pytorch.autoencoder.HeAE`
`__init__()` (`alibi.explainers.anchors.text_samplers.Neighbors` method), 724
`method`), 647 `__init__()` (`alibi.models.pytorch.cfml_models.ADULTDecoder`
`__init__()` (`alibi.explainers.anchors.text_samplers.SimilaritySampler` method), 724
`method`), 648 `__init__()` (`alibi.models.pytorch.cfml_models.ADULTEncoder`
`__init__()` (`alibi.explainers.anchors.text_samplers.UnknownSampler` method), 725
`method`), 649 `__init__()` (`alibi.models.pytorch.cfml_models.MNISTClassifier`
`__init__()` (`alibi.explainers.backends.pytorch.cfml_base.PtCounterfactualRLDataset`
`method`), 650 `__init__()` (`alibi.models.pytorch.cfml_models.MNISTDecoder`
`__init__()` (`alibi.explainers.backends.tensorflow.cfml_base.TfCounterfactualRLDataset`
`method`), 657 `__init__()` (`alibi.models.pytorch.cfml_models.MNISTEncoder`
`__init__()` (`alibi.explainers.cem.CEM` method), 682 `method`), 726
`__init__()` (`alibi.explainers.cfproto.CounterfactualProto` method), 685 `__init__()` (`alibi.models.pytorch.metrics.LossContainer`
`method`), 727
`__init__()` (`alibi.explainers.cfml_base.CounterfactualRL` method), 690 `__init__()` (`alibi.models.pytorch.metrics.Metric`
`method`), 728
`__init__()` (`alibi.explainers.cfml_base.NormalActionNoise` method), 693 `__init__()` (`alibi.models.tensorflow.actor_critic.Actor`
`method`), 730
`__init__()` (`alibi.explainers.cfml_base.ReplayBuffer` method), 694 `__init__()` (`alibi.models.tensorflow.actor_critic.Critic`
`method`), 731
`__init__()` (`alibi.explainers.cfml_tabular.CounterfactualRLTabular` method), 695 `__init__()` (`alibi.models.tensorflow.autoencoder.AE`
`method`), 732
`__init__()` (`alibi.explainers.cfml_tabular.SampleTabularPostprocessing` method), 698 `__init__()` (`alibi.models.tensorflow.autoencoder.HeAE`
`method`), 732
`__init__()` (`alibi.explainers.counterfactual.Counterfactual` method), 698 `__init__()` (`alibi.models.tensorflow.cfml_models.ADULTDecoder`
`method`), 733
`__init__()` (`alibi.explainers.integrated_gradients.IntegratedGradients` method), 700 `__init__()` (`alibi.models.tensorflow.cfml_models.ADULTEncoder`
`method`), 733
`__init__()` (`alibi.explainers.partial_dependence.PartialDependence` method), 702 `__init__()` (`alibi.models.tensorflow.cfml_models.MNISTClassifier`
`method`), 734
`__init__()` (`alibi.explainers.partial_dependence.PartialDependenceBase` method), 703 `__init__()` (`alibi.models.tensorflow.cfml_models.MNISTDecoder`
`method`), 734
`__init__()` (`alibi.explainers.partial_dependence.TreePartialDependence` method), 705 `__init__()` (`alibi.models.tensorflow.cfml_models.MNISTEncoder`
`method`), 735
`__init__()` (`alibi.explainers.pd_variance.PartialDependenceVariance` method), 708 `__init__()` (`alibi.prototypes.ProtoSelect` method), 735
`method`), 708 `__init__()` (`alibi.prototypes.protoselect.ProtoSelect`
`__init__()` (`alibi.explainers.permutation_importance.PermutationImportance` method), 712
`method`), 712 `__init__()` (`alibi.tests.utils.MockPredictor` method),
`__init__()` (`alibi.explainers.shap_wrappers.KernelExplainerWrapper` method), 715
`method`), 715 `__init__()` (`alibi.utils.BertBaseUncased` method), 742
`__init__()` (`alibi.explainers.shap_wrappers.KernelShap` method), 716 `__init__()` (`alibi.utils.DistilbertBaseUncased` method),
`method`), 716 742
`__init__()` (`alibi.explainers.shap_wrappers.TreeShap` method), 717 `__init__()` (`alibi.utils.DistributedExplainer` method),

- 743
 __init__() (alibi.utils.LanguageModel method), 745
 __init__() (alibi.utils.RobertaBase method), 748
 __init__() (alibi.utils.discretizer.Discretizer method), 752
 __init__() (alibi.utils.distributed.ActorPool method), 756
 __init__() (alibi.utils.distributed.DistributedExplainer method), 759
 __init__() (alibi.utils.distributed.PoolCollection method), 761
 __init__() (alibi.utils.kernel.EuclideanDistance method), 766
 __init__() (alibi.utils.kernel.GaussianRBF method), 766
 __init__() (alibi.utils.kernel.GaussianRBFDistance method), 767
 __init__() (alibi.utils.lang_model.BertBaseUncased method), 767
 __init__() (alibi.utils.lang_model.DistilbertBaseUncased method), 768
 __init__() (alibi.utils.lang_model.LanguageModel method), 769
 __init__() (alibi.utils.lang_model.RobertaBase method), 771
 __init__() (alibi.utils.missing_optional_dependency.MissingOptionalDependency method), 774
- ## A
- abdm() (in module alibi.utils.distance), 753
 absolute_value (alibi.utils.visualization.VisualizeSign attribute), 775
 AccuracyMetric (class in alibi.models.pytorch.metrics), 727
 Actor (class in alibi.models.pytorch.actor_critic), 722
 Actor (class in alibi.models.tensorflow.actor_critic), 730
 actor_index (alibi.utils.distributed.DistributedExplainer property), 760
 actor_index (alibi.utils.DistributedExplainer property), 744
 ActorPool (class in alibi.utils.distributed), 756
 adaptive_grid() (in module alibi.explainers.ale), 680
 add_names_to_exp() (alibi.explainers.anchors.anchor_tabular.AnchorTabular method), 630
 add_names_to_exp() (alibi.explainers.AnchorTabular method), 581
 add_noise() (in module alibi.explainers.backends.pytorch.cfrl_base), 651
 add_noise() (in module alibi.explainers.backends.tensorflow.cfrl_base), 657
 ADULTDecoder (class in alibi.models.pytorch.cfrl_models), 724
 ADULTDecoder (class in alibi.models.tensorflow.cfrl_models), 733
 ADULTEncoder (class in alibi.models.pytorch.cfrl_models), 725
 ADULTEncoder (class in alibi.models.tensorflow.cfrl_models), 733
 AE (class in alibi.models.pytorch.autoencoder), 723
 AE (class in alibi.models.tensorflow.autoencoder), 731
 ALE (class in alibi.explainers), 576
 ALE (class in alibi.explainers.ale), 678
 ale_num() (in module alibi.explainers.ale), 680
 alibi
 module, 561
 alibi.api
 module, 561
 alibi.api.defaults
 module, 561
 alibi.api.interfaces
 module, 564
 alibi.confidence
 module, 566
 alibi.confidence.model_linearity
 module, 569
 alibi.confidence.trustscore
 module, 570
 alibi.datasets
 module, 572
 alibi.datasets.default
 module, 574
 alibi.datasets.tensorflow
 module, 576
 alibi.exceptions
 module, 779
 alibi.explainers
 module, 576
 alibi.explainers.ale
 module, 678
 alibi.explainers.anchors
 module, 619
 alibi.explainers.anchors.anchor_base
 module, 619
 alibi.explainers.anchors.anchor_explanation
 module, 624
 alibi.explainers.anchors.anchor_image
 module, 625
 alibi.explainers.anchors.anchor_tabular
 module, 630
 alibi.explainers.anchors.anchor_tabular_distributed
 module, 637
 alibi.explainers.anchors.anchor_text
 module, 640
 alibi.explainers.anchors.language_model_text_sampler

- module, 644
- alibi.explainers.anchors.text_samplers
 - module, 647
- alibi.explainers.backends
 - module, 650
- alibi.explainers.backends.cfrl_base
 - module, 664
- alibi.explainers.backends.cfrl_tabular
 - module, 665
- alibi.explainers.backends.pytorch
 - module, 650
- alibi.explainers.backends.pytorch.cfrl_base
 - module, 650
- alibi.explainers.backends.pytorch.cfrl_tabular
 - module, 655
- alibi.explainers.backends.tensorflow
 - module, 657
- alibi.explainers.backends.tensorflow.cfrl_base
 - module, 657
- alibi.explainers.backends.tensorflow.cfrl_tabular
 - module, 663
- alibi.explainers.cem
 - module, 682
- alibi.explainers.cfproto
 - module, 685
- alibi.explainers.cfrl_base
 - module, 689
- alibi.explainers.cfrl_tabular
 - module, 695
- alibi.explainers.counterfactual
 - module, 698
- alibi.explainers.integrated_gradients
 - module, 700
- alibi.explainers.partial_dependence
 - module, 702
- alibi.explainers.pd_variance
 - module, 708
- alibi.explainers.permutation_importance
 - module, 711
- alibi.explainers.shap_wrappers
 - module, 715
- alibi.explainers.similarity
 - module, 673
- alibi.explainers.similarity.backends
 - module, 673
- alibi.explainers.similarity.backends.pytorch
 - module, 673
- alibi.explainers.similarity.backends.pytorch.base
 - module, 673
- alibi.explainers.similarity.backends.tensorflow
 - module, 673
- alibi.explainers.similarity.backends.tensorflow.base
 - module, 673
- alibi.explainers.similarity.base
 - module, 674
- alibi.explainers.similarity.grad
 - module, 675
- alibi.explainers.similarity.metrics
 - module, 677
- alibi.models
 - module, 722
- alibi.models.pytorch
 - module, 722
- alibi.models.pytorch.actor_critic
 - module, 722
- alibi.models.pytorch.autoencoder
 - module, 723
- alibi.models.pytorch.cfrl_models
 - module, 724
- alibi.models.pytorch.metrics
 - module, 727
- alibi.models.pytorch.model
 - module, 728
- alibi.models.tensorflow
 - module, 730
- alibi.models.tensorflow.actor_critic
 - module, 730
- alibi.models.tensorflow.autoencoder
 - module, 731
- alibi.models.tensorflow.cfrl_models
 - module, 733
- alibi.prototypes
 - module, 735
- alibi.prototypes.protoselect
 - module, 737
- alibi.saving
 - module, 779
- alibi.tests
 - module, 741
- alibi.tests.utils
 - module, 741
- alibi.utils
 - module, 742
- alibi.utils.approximation_methods
 - module, 751
- alibi.utils.data
 - module, 752
- alibi.utils.discretizer
 - module, 752
- alibi.utils.distance
 - module, 753
- alibi.utils.distributed
 - module, 756
- alibi.utils.distributions
 - module, 764
- alibi.utils.download
 - module, 765
- alibi.utils.frameworks

- module, 765
 - alibi.utils.gradients
 - module, 765
 - alibi.utils.kernel
 - module, 766
 - alibi.utils.lang_model
 - module, 767
 - alibi.utils.mapping
 - module, 772
 - alibi.utils.missing_optional_dependency
 - module, 774
 - alibi.utils.tf
 - module, 775
 - alibi.utils.visualization
 - module, 775
 - alibi.utils.wrappers
 - module, 778
 - alibi.version
 - module, 780
 - AlibiException, 779
 - AlibiPredictorCallException (class in alibi.exceptions), 779
 - AlibiPredictorReturnTypeError (class in alibi.exceptions), 779
 - AlibiPrettyPrinter (class in alibi.api.interfaces), 564
 - all (alibi.utils.visualization.VisualizeSign attribute), 775
 - alpha_scaling (alibi.utils.visualization.ImageVisualizationMethod attribute), 775
 - anchor_beam() (alibi.explainers.anchors.anchor_base.AnchorBaseBeam method), 619
 - AnchorBaseBeam (class in alibi.explainers.anchors.anchor_base), 619
 - AnchorExplanation (class in alibi.explainers.anchors.anchor_explanation), 624
 - AnchorImage (class in alibi.explainers), 578
 - AnchorImage (class in alibi.explainers.anchors.anchor_image), 625
 - AnchorImageSampler (class in alibi.explainers.anchors.anchor_image), 628
 - AnchorTabular (class in alibi.explainers), 581
 - AnchorTabular (class in alibi.explainers.anchors.anchor_tabular), 630
 - AnchorText (class in alibi.explainers), 583
 - AnchorText (class in alibi.explainers.anchors.anchor_text), 640
 - AnchorTextSampler (class in alibi.explainers.anchors.text_samplers), 647
 - append() (alibi.explainers.cfml_base.ReplayBuffer method), 694
 - apply_category_mapping() (in module alibi.explainers.backends.cfml_tabular), 665
 - approximation_parameters() (in module alibi.utils.approximation_methods), 751
 - argmax_grad() (in module alibi.utils.tf), 775
 - ArgmaxTransformer (class in alibi.utils.wrappers), 778
 - argmin_grad() (in module alibi.utils.tf), 775
 - assert_message_in_logs() (in module alibi.tests.utils), 741
 - asym_dot() (in module alibi.explainers.similarity.metrics), 677
 - attack() (alibi.explainers.CEM method), 588
 - attack() (alibi.explainers.cem.CEM method), 683
 - attack() (alibi.explainers.cfproto.CounterfactualProto method), 686
 - attack() (alibi.explainers.CounterfactualProto method), 592
 - AVERAGE (alibi.explainers.partial_dependence.Kind attribute), 702
- ## B
- Base (class in alibi.api.interfaces), 564
 - BaseSimilarityExplainer (class in alibi.explainers.similarity.base), 674
 - batch() (in module alibi.utils.distributed), 762
 - batch_compute_kernel_matrix() (in module alibi.utils.distance), 754
 - BertBaseUncased (class in alibi.utils), 742
 - BertBaseUncased (class in alibi.utils.lang_model), 767
 - bins() (alibi.utils.discretizer.Discretizer method), 753
 - bootstrap() (in module alibi.explainers.ale), 681
 - blended_heat_map (alibi.utils.visualization.ImageVisualizationMethod attribute), 775
 - BOTH (alibi.explainers.partial_dependence.Kind attribute), 702
 - build() (alibi.models.tensorflow.autoencoder.HeAE method), 732
 - build_lookups() (alibi.explainers.anchors.anchor_tabular.TabularSampler method), 634
 - build_lookups() (alibi.explainers.anchors.anchor_tabular_distributed.RemoteSampler method), 639
 - Bunch (class in alibi.utils.data), 752
- ## C
- call() (alibi.models.tensorflow.actor_critic.Actor method), 731
 - call() (alibi.models.tensorflow.actor_critic.Critic method), 731
 - call() (alibi.models.tensorflow.autoencoder.AE method), 732
 - call() (alibi.models.tensorflow.autoencoder.HeAE method), 732

`call()` (*alibi.models.tensorflow.cfml_models.ADULTDecoder* method), 733
`call()` (*alibi.models.tensorflow.cfml_models.ADULTEncoder* method), 733
`call()` (*alibi.models.tensorflow.cfml_models.MNISTClassifier* method), 734
`call()` (*alibi.models.tensorflow.cfml_models.MNISTDecoder* method), 734
`call()` (*alibi.models.tensorflow.cfml_models.MNISTEncoder* method), 735
`CALLABLE` (*alibi.explainers.integrated_gradients.LayerState* attribute), 701
`Callback` (class in *alibi.explainers.cfml_base*), 689
`caller` (*alibi.utils.lang_model.BertBaseUncased* attribute), 767
`caller` (*alibi.utils.lang_model.DistilbertBaseUncased* attribute), 768
`caller` (*alibi.utils.lang_model.LanguageModel* attribute), 769
`caller` (*alibi.utils.lang_model.RobertaBase* attribute), 771
`caller` (*alibi.utils.LanguageModel* attribute), 745
`CEM` (class in *alibi.explainers*), 587
`CEM` (class in *alibi.explainers.cem*), 682
`cityblock_batch()` (in module *alibi.utils.distance*), 754
`CLASS_SAMPLER` (*alibi.explainers.anchors.anchor_text.AnchorText* attribute), 640
`CLASS_SAMPLER` (*alibi.explainers.AnchorText* attribute), 583
`CLASSIFICATION` (*alibi.explainers.similarity.grad.Task* attribute), 677
`compare_labels()` (*alibi.explainers.anchors.anchor_image.AnchorImage* method), 629
`compare_labels()` (*alibi.explainers.anchors.anchor_tabular.TabularSampler* method), 634
`compare_labels()` (*alibi.explainers.anchors.anchor_text.AnchorText* method), 641
`compare_labels()` (*alibi.explainers.AnchorText* method), 584
`compile()` (*alibi.models.pytorch.model.Model* method), 728
`compute_beta()` (*alibi.explainers.anchors.anchor_base.AnchorBase* static method), 620
`compute_loss()` (*alibi.models.pytorch.model.Model* method), 729
`compute_metric()` (*alibi.models.pytorch.metrics.AccuracyMetric* method), 727
`compute_metric()` (*alibi.models.pytorch.metrics.Metric* method), 728
`compute_metrics()` (*alibi.models.pytorch.model.Model* method), 729
`compute_prototype_importances()` (in module *alibi.prototypes.protoselect*), 738
`concatenate` (*alibi.utils.distributed.DistributedExplainer* attribute), 760
`concatenate` (*alibi.utils.DistributedExplainer* attribute), 744
`concatenate_minibatches()` (in module *alibi.utils.distributed*), 763
`ConcatTabularPostprocessing` (class in *alibi.explainers.cfml_tabular*), 695
`consistency_loss()` (in module *alibi.explainers.backends.pytorch.cfml_base*), 651
`consistency_loss()` (in module *alibi.explainers.backends.pytorch.cfml_tabular*), 655
`consistency_loss()` (in module *alibi.explainers.backends.tensorflow.cfml_base*), 658
`consistency_loss()` (in module *alibi.explainers.backends.tensorflow.cfml_tabular*), 663
`cosine()` (in module *alibi.explainers.similarity.metrics*), 677
`Counterfactual` (class in *alibi.explainers*), 589
`Counterfactual` (class in *alibi.explainers.counterfactual*), 698
`CounterFactual()` (in module *alibi.explainers.counterfactual*), 698
`CounterfactualProto` (class in *alibi.explainers*), 591
`CounterfactualProto` (class in *alibi.explainers.cfproto*), 685
`CounterFactualProto()` (in module *alibi.explainers.cfproto*), 685
`CounterfactualRL` (class in *alibi.explainers*), 595
`CounterfactualRL` (class in *alibi.explainers.cfml_base*), 690
`CounterfactualRLDataset` (class in *alibi.explainers.backends.cfml_base*), 664
`CounterfactualRLTabular` (class in *alibi.explainers*), 597
`CounterfactualRLTabular` (class in *alibi.explainers.cfml_tabular*), 695
`coverage()` (*alibi.explainers.anchors.anchor_explanation.AnchorExplanation* method), 624
`create_explainer_handles()` (*alibi.utils.distributed.PoolCollection* static method), 762
`create_mask()` (*alibi.explainers.anchors.language_model_text_sampler.LanguageModelTextSampler* method), 645
`create_parallel_pool()` (*alibi.utils.distributed.PoolCollection* static method), 762

- ibi.utils.distributed.DistributedExplainer* method), 760
- `create_parallel_pool()` (*alibi.utils.DistributedExplainer* method), 744
- `Critic` (class in *alibi.models.pytorch.actor_critic*), 722
- `Critic` (class in *alibi.models.tensorflow.actor_critic*), 731
- `cv_protoselect_euclidean()` (in module *alibi.prototypes.protoselect*), 739
- ## D
- `data` (*alibi.api.interfaces.Explanation* attribute), 565
- `data_generator()` (in module *alibi.explainers.backends.pytorch.cfrl_base*), 651
- `data_generator()` (in module *alibi.explainers.backends.tensorflow.cfrl_base*), 658
- `decode()` (in module *alibi.explainers.backends.pytorch.cfrl_base*), 652
- `decode()` (in module *alibi.explainers.backends.tensorflow.cfrl_base*), 658
- `default()` (*alibi.saving.NumpyEncoder* method), 779
- `DEFAULT_BASE_PARAMS` (in module *alibi.explainers.cfrl_base*), 691
- `DEFAULT_DATA_ALE` (in module *alibi.api.defaults*), 561
- `DEFAULT_DATA_ANCHOR` (in module *alibi.api.defaults*), 561
- `DEFAULT_DATA_ANCHOR_IMG` (in module *alibi.api.defaults*), 561
- `DEFAULT_DATA_CEM` (in module *alibi.api.defaults*), 561
- `DEFAULT_DATA_CF` (in module *alibi.api.defaults*), 561
- `DEFAULT_DATA_CFP` (in module *alibi.api.defaults*), 561
- `DEFAULT_DATA_CFRL` (in module *alibi.api.defaults*), 561
- `DEFAULT_DATA_INTGRAD` (in module *alibi.api.defaults*), 562
- `DEFAULT_DATA_KERNEL_SHAP` (in module *alibi.api.defaults*), 562
- `DEFAULT_DATA_PD` (in module *alibi.api.defaults*), 562
- `DEFAULT_DATA_PDVARIANCE` (in module *alibi.api.defaults*), 562
- `DEFAULT_DATA_PERMUTATION_IMPORTANCE` (in module *alibi.api.defaults*), 562
- `DEFAULT_DATA_PROTOSELECT` (in module *alibi.api.defaults*), 562
- `DEFAULT_DATA_SIM` (in module *alibi.api.defaults*), 562
- `DEFAULT_DATA_TREE_SHAP` (in module *alibi.api.defaults*), 562
- `default_meta()` (in module *alibi.api.interfaces*), 565
- `DEFAULT_META_ALE` (in module *alibi.api.defaults*), 562
- `DEFAULT_META_ANCHOR` (in module *alibi.api.defaults*), 562
- `DEFAULT_META_CEM` (in module *alibi.api.defaults*), 562
- `DEFAULT_META_CF` (in module *alibi.api.defaults*), 562
- `DEFAULT_META_CFP` (in module *alibi.api.defaults*), 562
- `DEFAULT_META_CFRL` (in module *alibi.api.defaults*), 563
- `DEFAULT_META_INTGRAD` (in module *alibi.api.defaults*), 563
- `DEFAULT_META_KERNEL_SHAP` (in module *alibi.api.defaults*), 563
- `DEFAULT_META_PD` (in module *alibi.api.defaults*), 563
- `DEFAULT_META_PDVARIANCE` (in module *alibi.api.defaults*), 563
- `DEFAULT_META_PERMUTATION_IMPORTANCE` (in module *alibi.api.defaults*), 563
- `DEFAULT_META_PROTOSELECT` (in module *alibi.api.defaults*), 563
- `DEFAULT_META_SIM` (in module *alibi.api.defaults*), 563
- `DEFAULT_META_TREE_SHAP` (in module *alibi.api.defaults*), 563
- `DEFAULT_SAMPLING_LANGUAGE_MODEL` (in module *alibi.explainers.anchors.anchor_text*), 643
- `DEFAULT_SAMPLING_SIMILARITY` (in module *alibi.explainers.anchors.anchor_text*), 644
- `DEFAULT_SAMPLING_UNKNOWN` (in module *alibi.explainers.anchors.anchor_text*), 644
- `default_target_fcn()` (in module *alibi.utils.distributed*), 763
- `DEFAULTS` (*alibi.explainers.anchors.anchor_text.AnchorText* attribute), 640
- `DEFAULTS` (*alibi.explainers.AnchorText* attribute), 583
- `deferred_init()` (*alibi.explainers.anchors.anchor_tabular.TabularSampler* method), 634
- `DIFFERENCE` (*alibi.explainers.permutation_importance.Kind* attribute), 711
- `discretize()` (*alibi.utils.discretizer.Discretizer* method), 753
- `Discretizer` (class in *alibi.utils.discretizer*), 752
- `DistilbertBaseUncased` (class in *alibi.utils*), 742
- `DistilbertBaseUncased` (class in *alibi.utils.lang_model*), 768
- `DISTRIBUTED_OPTS` (in module *alibi.explainers.shap_wrappers*), 715
- `DistributedAnchorBaseBeam` (class in *alibi.explainers.anchors.anchor_tabular_distributed*), 637
- `DistributedAnchorTabular` (class in *alibi.explainers*), 599
- `DistributedAnchorTabular` (class in *alibi.explainers.anchors.anchor_tabular_distributed*), 637
- `DistributedExplainer` (class in *alibi.utils*), 743
- `DistributedExplainer` (class in *alibi.utils.distributed*), 758
- `dlow_bernoulli()` (al-

`ibi.explainers.anchors.anchor_base.AnchorBaseBeam`.`explain()` (`alibi.explainers.Counterfactual` method), static method), 620 590
`dot()` (in module `alibi.explainers.similarity.metrics`), 677 `explain()` (`alibi.explainers.counterfactual.Counterfactual` method), 699
`draw_samples()` (`alibi.explainers.anchors.anchor_base.AnchorBaseBeam` method), 621 `explain()` (`alibi.explainers.CounterfactualProto` method), 593
`draw_samples()` (`alibi.explainers.anchors.anchor_tabular_distributed.DistributedAnchorBaseBeam` method), 637 `explain()` (`alibi.explainers.CounterfactualRL` method), 596
`dup_bernoulli()` (`al-` 596
`ibi.explainers.anchors.anchor_base.AnchorBaseBeam`.`explain()` (`alibi.explainers.CounterfactualRLTabular` static method), 621 method), 598
E
`encode()` (in module `al-` `explain()` (`alibi.explainers.GradientSimilarity` method), 601
`ibi.explainers.backends.pytorch.cfml_base`), 652 `explain()` (`alibi.explainers.integrated_gradients.IntegratedGradients` method), 700
`encode()` (in module `al-` `explain()` (`alibi.explainers.IntegratedGradients` method), 603
`ibi.explainers.backends.tensorflow.cfml_base`), 659
`err_msg` (`alibi.utils.missing_optional_dependency.MissingDependency` property), 774 `explain()` (`alibi.explainers.KernelShap` method), 604
`err_msg_template` (in module `al-` `explain()` (`alibi.explainers.partial_dependence.PartialDependence` method), 702
`ibi.utils.missing_optional_dependency`), 774 `explain()` (`alibi.explainers.partial_dependence.PartialDependenceBase` method), 704
`ESTIMATE` (`alibi.explainers.permutation_importance.Method` attribute), 711 `explain()` (`alibi.explainers.partial_dependence.TreePartialDependence` method), 706
`EuclideanDistance` (class in `alibi.utils.kernel`), 766
`evaluate()` (`alibi.models.pytorch.model.Model` method), 729 `explain()` (`alibi.explainers.PartialDependence` method), 607
`EXACT` (`alibi.explainers.permutation_importance.Method` attribute), 712 `explain()` (`alibi.explainers.PartialDependenceVariance` method), 609
`examples()` (`alibi.explainers.anchors.anchor_explanation.AnchorExplanation` method), 624 `explain()` (`alibi.explainers.pd_variance.PartialDependenceVariance` method), 709
`explain()` (`alibi.api.interfaces.Explainer` method), 564 `explain()` (`alibi.explainers.permutation_importance.PermutationImportance` method), 713
`explain()` (`alibi.explainers.ALE` method), 577 `explain()` (`alibi.explainers.PermutationImportance` method), 611
`explain()` (`alibi.explainers.ale.ALE` method), 679 `explain()` (`alibi.explainers.shap_wrappers.KernelShap` method), 716
`explain()` (`alibi.explainers.anchors.anchor_image.AnchorImage` method), 578 `explain()` (`alibi.explainers.shap_wrappers.TreeShap` method), 719
`explain()` (`alibi.explainers.anchors.anchor_image.AnchorImage` method), 626 `explain()` (`alibi.explainers.similarity.grad.GradientSimilarity` method), 676
`explain()` (`alibi.explainers.anchors.anchor_tabular.AnchorTabular` method), 631 `explain()` (`alibi.explainers.TreePartialDependence` method), 612
`explain()` (`alibi.explainers.anchors.anchor_tabular_distributed.DistributedAnchorTabular` method), 637 `explain()` (`alibi.explainers.TreeShap` method), 614
`explain()` (`alibi.explainers.anchors.anchor_text.AnchorText` method), 641 `Explainer` (class in `alibi.api.interfaces`), 564
`explain()` (`alibi.explainers.AnchorTabular` method), 581 `Explanation` (class in `alibi.api.interfaces`), 564
`explain()` (`alibi.explainers.AnchorText` method), 584
`explain()` (`alibi.explainers.CEM` method), 588
`explain()` (`alibi.explainers.cem.CEM` method), 683
`explain()` (`alibi.explainers.cfproto.CounterfactualProto` method), 687
`explain()` (`alibi.explainers.cfml_base.CounterfactualRL` method), 690
`explain()` (`alibi.explainers.cfml_tabular.CounterfactualRLTabular` method), 696
F
`features()` (`alibi.explainers.anchors.anchor_explanation.AnchorExplanation` method), 624
`fetch_adult()` (in module `alibi.datasets`), 572
`fetch_adult()` (in module `alibi.datasets.default`), 574
`fetch_fashion_mnist()` (in module `alibi.datasets`), 572

- [fetch_fashion_mnist\(\)](#) (in module `alibi.datasets.tensorflow`), 576
[fetch_imagenet\(\)](#) (in module `alibi.datasets`), 572
[fetch_imagenet\(\)](#) (in module `alibi.datasets.default`), 574
[fetch_imagenet_10\(\)](#) (in module `alibi.datasets`), 572
[fetch_imagenet_10\(\)](#) (in module `alibi.datasets.default`), 574
[fetch_movie_sentiment\(\)](#) (in module `alibi.datasets`), 573
[fetch_movie_sentiment\(\)](#) (in module `alibi.datasets.default`), 575
[fill_mask\(\)](#) (`alibi.explainers.anchors.language_model_text_sampler` method), 645
[FILLING_AUTOREGRESSIVE](#) (`alibi.explainers.anchors.language_model_text_sampler` attribute), 644
[FILLING_PARALLEL](#) (`alibi.explainers.anchors.language_model_text_sampler` attribute), 644
[filter_by_distance_knn\(\)](#) (`alibi.confidence.TrustScore` method), 567
[filter_by_distance_knn\(\)](#) (`alibi.confidence.trustscore.TrustScore` method), 571
[filter_by_probability_knn\(\)](#) (`alibi.confidence.TrustScore` method), 567
[filter_by_probability_knn\(\)](#) (`alibi.confidence.trustscore.TrustScore` method), 571
[find_similar_words\(\)](#) (`alibi.explainers.anchors.text_samplers.SimilaritySampler` method), 648
[fit\(\)](#) (`alibi.api.interfaces.FitMixin` method), 565
[fit\(\)](#) (`alibi.confidence.LinearityMeasure` method), 566
[fit\(\)](#) (`alibi.confidence.model_linearity.LinearityMeasure` method), 569
[fit\(\)](#) (`alibi.confidence.TrustScore` method), 567
[fit\(\)](#) (`alibi.confidence.trustscore.TrustScore` method), 571
[fit\(\)](#) (`alibi.explainers.anchors.anchor_tabular.AnchorTabular` method), 632
[fit\(\)](#) (`alibi.explainers.anchors.anchor_tabular_distributed.DistributedAnchorTabular` method), 638
[fit\(\)](#) (`alibi.explainers.AnchorTabular` method), 583
[fit\(\)](#) (`alibi.explainers.CEM` method), 588
[fit\(\)](#) (`alibi.explainers.cem.CEM` method), 684
[fit\(\)](#) (`alibi.explainers.cfproto.CounterfactualProto` method), 687
[fit\(\)](#) (`alibi.explainers.cfml_base.CounterfactualRL` method), 691
[fit\(\)](#) (`alibi.explainers.cfml_tabular.CounterfactualRLTabular` method), 697
[fit\(\)](#) (`alibi.explainers.Counterfactual` method), 591
[fit\(\)](#) (`alibi.explainers.counterfactual.Counterfactual` method), 699
[fit\(\)](#) (`alibi.explainers.CounterfactualProto` method), 593
[fit\(\)](#) (`alibi.explainers.CounterfactualRL` method), 596
[fit\(\)](#) (`alibi.explainers.CounterfactualRLTabular` method), 599
[fit\(\)](#) (`alibi.explainers.DistributedAnchorTabular` method), 600
[fit\(\)](#) (`alibi.explainers.GradientSimilarity` method), 602
[fit\(\)](#) (`alibi.explainers.KernelShap` method), 605
[fit\(\)](#) (`alibi.explainers.shap_wrappers.KernelShap` method), 605
[fit\(\)](#) (`alibi.explainers.shap_wrappers.TreeShap` method), 720
[fit\(\)](#) (`alibi.explainers.similarity.base.BaseSimilarityExplainer` method), 674
[fit\(\)](#) (`alibi.explainers.similarity.grad.GradientSimilarity` method), 615
[fit\(\)](#) (`alibi.explainers.TreeShap` method), 615
[fit\(\)](#) (`alibi.models.pytorch.model.Model` method), 729
[fit\(\)](#) (`alibi.prototypes.ProtoSelect` method), 736
[fit\(\)](#) (`alibi.prototypes.prototype.ProtoSelect` method), 738
[FitMixin](#) (class in `alibi.api.interfaces`), 565
[forward\(\)](#) (`alibi.models.pytorch.actor_critic.Actor` method), 722
[forward\(\)](#) (`alibi.models.pytorch.actor_critic.Critic` method), 723
[forward\(\)](#) (`alibi.models.pytorch.autoencoder.AE` method), 723
[forward\(\)](#) (`alibi.models.pytorch.autoencoder.HeAE` method), 724
[forward\(\)](#) (`alibi.models.pytorch.cfml_models.ADULTDecoder` method), 724
[forward\(\)](#) (`alibi.models.pytorch.cfml_models.ADULTEncoder` method), 725
[forward\(\)](#) (`alibi.models.pytorch.cfml_models.MNISTClassifier` method), 725
[forward\(\)](#) (`alibi.models.pytorch.cfml_models.MNISTDecoder` method), 726
[forward\(\)](#) (`alibi.models.pytorch.cfml_models.MNISTEncoder` method), 726
[Framework](#) (class in `alibi.utils.frameworks`), 765
[from_disk\(\)](#) (`alibi.utils.lang_model.LanguageModel` method), 769
[from_disk\(\)](#) (`alibi.utils.LanguageModel` method), 745
[from_json\(\)](#) (`alibi.api.interfaces.Explanation` class method), 565
- ## G
- [gauss_legendre_builders\(\)](#) (in module `alibi.utils.approximation_methods`), 751
[GaussianRBF](#) (class in `alibi.utils.kernel`), 766

GaussianRBFDistance (class in *alibi.utils.kernel*), 767
 gen_category_map() (in module *alibi.utils*), 748
 gen_category_map() (in module *alibi.utils.data*), 752
 generate_categorical_condition() (in module *alibi.explainers.backends.cfml_tabular*), 666
 generate_cf() (in module *alibi.explainers.backends.pytorch.cfml_base*), 652
 generate_cf() (in module *alibi.explainers.backends.tensorflow.cfml_base*), 659
 generate_condition() (in module *alibi.explainers.backends.cfml_tabular*), 666
 generate_empty_condition() (in module *alibi.explainers.backends.cfml_base*), 664
 generate_numerical_condition() (in module *alibi.explainers.backends.cfml_tabular*), 667
 generate_superpixels() (*alibi.explainers.AnchorImage* method), 580
 generate_superpixels() (*alibi.explainers.anchors.anchor_image.AnchorImage* method), 627
 generate_superpixels() (*alibi.explainers.anchors.anchor_image.AnchorImageSampler* method), 629
 get_actor() (in module *alibi.explainers.backends.pytorch.cfml_base*), 653
 get_actor() (in module *alibi.explainers.backends.tensorflow.cfml_base*), 659
 get_anchor_metadata() (*alibi.explainers.anchors.anchor_base.AnchorBaseBeam* method), 621
 get_categorical_conditional_vector() (in module *alibi.explainers.backends.cfml_tabular*), 667
 get_classification_reward() (in module *alibi.explainers.backends.cfml_base*), 665
 get_conditional_dim() (in module *alibi.explainers.backends.cfml_tabular*), 668
 get_conditional_vector() (in module *alibi.explainers.backends.cfml_tabular*), 668
 get_critic() (in module *alibi.explainers.backends.pytorch.cfml_base*), 653
 get_critic() (in module *alibi.explainers.backends.tensorflow.cfml_base*), 659
 get_device() (in module *alibi.explainers.backends.pytorch.cfml_base*), 653
 get_explanation() (*alibi.explainers.shap_wrappers.KernelExplainerWrapper* method), 715
 get_explanation() (*alibi.utils.distributed.DistributedExplainer* method), 760
 get_explanation() (*alibi.utils.distributed.PoolCollection* method), 762
 get_explanation() (*alibi.utils.DistributedExplainer* method), 744
 get_features_index() (*alibi.explainers.anchors.anchor_tabular.TabularSampler* method), 635
 get_gradients() (*alibi.explainers.CEM* method), 588
 get_gradients() (*alibi.explainers.cem.CEM* method), 684
 get_gradients() (*alibi.explainers.cfproto.CounterfactualProto* method), 688
 get_gradients() (*alibi.explainers.CounterfactualProto* method), 594
 get_hard_distribution() (in module *alibi.explainers.backends.cfml_base*), 665
 get_he_preprocessor() (in module *alibi.explainers.backends.cfml_tabular*), 669
 get_init_stats() (*alibi.explainers.anchors.anchor_base.AnchorBaseBeam* method), 621
 get_next() (*alibi.utils.distributed.ActorPool* method), 756
 get_next_unordered() (*alibi.utils.distributed.ActorPool* method), 756
 get_numerical_conditional_vector() (in module *alibi.explainers.backends.cfml_tabular*), 670
 get_optimizer() (in module *alibi.explainers.backends.pytorch.cfml_base*), 653
 get_optimizer() (in module *alibi.explainers.backends.tensorflow.cfml_base*), 660
 get_percentiles() (*alibi.utils.discretizer.Discretizer* static method), 753
 get_quantiles() (in module *alibi.explainers.ale*), 681
 get_sample_ids() (*alibi.explainers.anchors.language_model_text_sampler.LanguageModelTextSampler* method), 646
 get_statistics() (in module *alibi.explainers.backends.cfml_tabular*), 671
 GradientSimilarity (class in *alibi.explainers*), 600
 GradientSimilarity (class in *alibi.explainers.similarity.grad*), 675
 handle_unk_features() (*alibi.explainers.anchors.anchor_tabular.TabularSampler* method), 635

H

- method), 635
- has_next() (alibi.utils.distributed.ActorPool method), 757
- head_tail_split() (alibi.utils.lang_model.LanguageModel method), 769
- head_tail_split() (alibi.utils.LanguageModel method), 745
- HeAE (class in alibi.models.pytorch.autoencoder), 723
- HeAE (class in alibi.models.tensorflow.autoencoder), 732
- heat_map (alibi.utils.visualization.ImageVisualizationMethod attribute), 775
- heatmap() (in module alibi.utils.visualization), 775
- I
- identity_function() (in module alibi.explainers.backends.cfml_base), 665
- ImageVisualizationMethod (class in alibi.utils.visualization), 775
- import_optional() (in module alibi.utils.missing_optional_dependency), 774
- IMPORTANCE (alibi.explainers.pd_variance.Method attribute), 708
- INDIVIDUAL (alibi.explainers.partial_dependence.Kind attribute), 702
- infer_feature_range() (in module alibi.confidence.model_linearity), 569
- initialize_actor_critic() (in module alibi.explainers.backends.tensorflow.cfml_base), 660
- initialize_optimizer() (in module alibi.explainers.backends.tensorflow.cfml_base), 660
- initialize_optimizers() (in module alibi.explainers.backends.tensorflow.cfml_base), 660
- instance_label (alibi.explainers.anchors.anchor_tabular_kind.attribute), 632
- instance_label (alibi.explainers.anchors.anchor_tabular_kind.attribute), 635
- instance_label (alibi.explainers.AnchorTabular attribute), 583
- IntegratedGradients (class in alibi.explainers), 602
- IntegratedGradients (class in alibi.explainers.integrated_gradients), 700
- INTERACTION (alibi.explainers.pd_variance.Method attribute), 708
- invert_permutation() (in module alibi.utils.distributed), 764
- is_punctuation() (alibi.utils.lang_model.LanguageModel method), 769
- is_punctuation() (alibi.utils.LanguageModel method), 746
- is_stop_word() (alibi.utils.lang_model.LanguageModel method), 769
- is_stop_word() (alibi.utils.LanguageModel method), 746
- is_subword_prefix() (alibi.utils.BertBaseUncased method), 742
- is_subword_prefix() (alibi.utils.DistilbertBaseUncased method), 742
- is_subword_prefix() (alibi.utils.lang_model.BertBaseUncased method), 767
- is_subword_prefix() (alibi.utils.lang_model.DistilbertBaseUncased method), 768
- is_subword_prefix() (alibi.utils.lang_model.LanguageModel method), 770
- is_subword_prefix() (alibi.utils.lang_model.RobertaBase method), 771
- is_subword_prefix() (alibi.utils.LanguageModel method), 746
- is_subword_prefix() (alibi.utils.RobertaBase method), 748
- issorted() (in module alibi.tests.utils), 741
- K
- KERNEL_SHAP_PARAMS (in module alibi.api.defaults), 563
- KernelExplainerWrapper (class in alibi.explainers.shap_wrappers), 715
- KernelShap (class in alibi.explainers), 604
- KernelShap (class in alibi.explainers.shap_wrappers), 715
- Kind (class in alibi.explainers.partial_dependence), 702
- Kind (class in alibi.explainers.permutation_importance), 711
- KullbackLeiblerDivergence (in module alibi.utils.distributions), 764
- kllucb() (alibi.explainers.anchors.anchor_base.AnchorBaseBeam method), 622
- L
- l0_ohe() (in module alibi.explainers.backends.pytorch.cfml_tabular), 655
- l0_ohe() (in module alibi.explainers.backends.tensorflow.cfml_tabular), 663
- l1_loss() (in module alibi.explainers.backends.pytorch.cfml_tabular), 656

- l1_loss() (in module *alibi.explainers.backends.tensorflow.cfml_tabular*), 663
- LanguageModel (class in *alibi.utils*), 745
- LanguageModel (class in *alibi.utils.lang_model*), 769
- LanguageModelSampler (class in *alibi.explainers.anchors.language_model_text_sampler*), 644
- LayerState (class in *alibi.explainers.integrated_gradients*), 701
- left (*alibi.utils.approximation_methods.Riemann* attribute), 751
- linearity_measure() (in module *alibi.confidence*), 568
- linearity_measure() (in module *alibi.confidence.model_linearity*), 570
- LinearityMeasure (class in *alibi.confidence*), 566
- LinearityMeasure (class in *alibi.confidence.model_linearity*), 569
- load() (*alibi.api.interfaces.Explainer* class method), 564
- load() (*alibi.api.interfaces.Summariser* class method), 565
- load() (*alibi.explainers.cfml_base.CounterfactualRL* class method), 691
- load() (*alibi.explainers.CounterfactualRL* class method), 596
- load_cats() (in module *alibi.datasets*), 573
- load_cats() (in module *alibi.datasets.default*), 575
- load_explainer() (in module *alibi.saving*), 780
- load_model() (in module *alibi.explainers.backends.pytorch.cfml_base*), 653
- load_model() (in module *alibi.explainers.backends.tensorflow.cfml_base*), 661
- load_spacy_lexeme_prob() (in module *alibi.explainers.anchors.text_samplers*), 650
- load_weights() (*alibi.models.pytorch.model.Model* method), 729
- loss_fn() (*alibi.explainers.CEM* method), 589
- loss_fn() (*alibi.explainers.cem.CEM* method), 684
- loss_fn() (*alibi.explainers.cfproto.CounterfactualProto* method), 688
- loss_fn() (*alibi.explainers.CounterfactualProto* method), 594
- LOSS_FNS (in module *alibi.explainers.permutation_importance*), 711
- LossContainer (class in *alibi.models.pytorch.metrics*), 727
- M**
- map() (*alibi.utils.distributed.ActorPool* method), 757
- map_unordered() (*alibi.utils.distributed.ActorPool* method), 758
- mask (*alibi.utils.BertBaseUncased* property), 742
- mask (*alibi.utils.DistilbertBaseUncased* property), 743
- mask (*alibi.utils.lang_model.BertBaseUncased* property), 768
- mask (*alibi.utils.lang_model.DistilbertBaseUncased* property), 768
- mask (*alibi.utils.lang_model.LanguageModel* property), 770
- mask (*alibi.utils.lang_model.RobertaBase* property), 772
- mask (*alibi.utils.LanguageModel* property), 746
- mask (*alibi.utils.RobertaBase* property), 748
- mask_id (*alibi.utils.lang_model.LanguageModel* property), 770
- mask_id (*alibi.utils.LanguageModel* property), 747
- masked_image (*alibi.utils.visualization.ImageVisualizationMethod* attribute), 775
- max_num_tokens (*alibi.utils.lang_model.LanguageModel* property), 770
- max_num_tokens (*alibi.utils.LanguageModel* property), 747
- MEAN (*alibi.models.pytorch.metrics.Reduction* attribute), 728
- meta (*alibi.api.interfaces.Base* attribute), 564
- meta (*alibi.api.interfaces.Explanation* attribute), 565
- meta (*alibi.explainers.anchors.anchor_tabular.AnchorTabular* attribute), 632
- meta (*alibi.explainers.anchors.anchor_text.AnchorText* attribute), 642
- methdispatch() (in module *alibi.utils.wrappers*), 778
- Method (class in *alibi.explainers.pd_variance*), 708
- Method (class in *alibi.explainers.permutation_importance*), 711
- Metric (class in *alibi.models.pytorch.metrics*), 728
- middle (*alibi.utils.approximation_methods.Riemann* attribute), 751
- minimum_satisfied() (in module *alibi.explainers.ale*), 681
- MissingDependency (class in *alibi.utils.missing_optional_dependency*), 774
- MNISTClassifier (class in *alibi.models.pytorch.cfml_models*), 725
- MNISTClassifier (class in *alibi.models.tensorflow.cfml_models*), 734
- MNISTDecoder (class in *alibi.models.pytorch.cfml_models*), 725
- MNISTDecoder (class in *alibi.models.tensorflow.cfml_models*), 734
- MNISTEncoder (class in *alibi.models.pytorch.cfml_models*), 726
- MNISTEncoder (class in *alibi.models.tensorflow.cfml_models*), 735
- MockPredictor (class in *alibi.tests.utils*), 741

- `model` (*alibi.explainers.anchors.anchor_text.AnchorText* attribute), 642
- `model` (*alibi.explainers.AnchorText* attribute), 586
- `model` (*alibi.utils.lang_model.BertBaseUncased* attribute), 768
- `model` (*alibi.utils.lang_model.DistilbertBaseUncased* attribute), 769
- `model` (*alibi.utils.lang_model.LanguageModel* attribute), 770
- `model` (*alibi.utils.lang_model.RobertaBase* attribute), 772
- `model` (*alibi.utils.LanguageModel* attribute), 747
- `Model` (class in *alibi.models.pytorch.model*), 728
- module
 - alibi*, 561
 - alibi.api*, 561
 - alibi.api.defaults*, 561
 - alibi.api.interfaces*, 564
 - alibi.confidence*, 566
 - alibi.confidence.model_linearity*, 569
 - alibi.confidence.trustscore*, 570
 - alibi.datasets*, 572
 - alibi.datasets.default*, 574
 - alibi.datasets.tensorflow*, 576
 - alibi.exceptions*, 779
 - alibi.explainers*, 576
 - alibi.explainers.ale*, 678
 - alibi.explainers.anchors*, 619
 - alibi.explainers.anchors.anchor_base*, 619
 - alibi.explainers.anchors.anchor_explanation*, 624
 - alibi.explainers.anchors.anchor_image*, 625
 - alibi.explainers.anchors.anchor_tabular*, 630
 - alibi.explainers.anchors.anchor_tabular_distribution*, 637
 - alibi.explainers.anchors.anchor_text*, 640
 - alibi.explainers.anchors.language_model_text_sampler*, 644
 - alibi.explainers.anchors.text_samplers*, 647
 - alibi.explainers.backends*, 650
 - alibi.explainers.backends.cfml_base*, 664
 - alibi.explainers.backends.cfml_tabular*, 665
 - alibi.explainers.backends.pytorch*, 650
 - alibi.explainers.backends.pytorch.cfml_base*, 650
 - alibi.explainers.backends.pytorch.cfml_tabular*, 655
 - alibi.explainers.backends.tensorflow*, 657
 - alibi.explainers.backends.tensorflow.cfml_base*, 657
 - alibi.explainers.backends.tensorflow.cfml_tabular*, 663
 - alibi.explainers.cem*, 682
 - alibi.explainers.cfproto*, 685
 - alibi.explainers.cfml_base*, 689
 - alibi.explainers.cfml_tabular*, 695
 - alibi.explainers.counterfactual*, 698
 - alibi.explainers.integrated_gradients*, 700
 - alibi.explainers.partial_dependence*, 702
 - alibi.explainers.pd_variance*, 708
 - alibi.explainers.permutation_importance*, 711
 - alibi.explainers.shap_wrappers*, 715
 - alibi.explainers.similarity*, 673
 - alibi.explainers.similarity.backends*, 673
 - alibi.explainers.similarity.backends.pytorch*, 673
 - alibi.explainers.similarity.backends.pytorch.base*, 673
 - alibi.explainers.similarity.backends.tensorflow*, 673
 - alibi.explainers.similarity.backends.tensorflow.base*, 673
 - alibi.explainers.similarity.base*, 674
 - alibi.explainers.similarity.grad*, 675
 - alibi.explainers.similarity.metrics*, 677
 - alibi.models*, 722
 - alibi.models.pytorch*, 722
 - alibi.models.pytorch.actor_critic*, 722
 - alibi.models.pytorch.autoencoder*, 723
 - alibi.models.pytorch.cfml_models*, 724
 - alibi.models.pytorch.metrics*, 727
 - alibi.models.pytorch.model*, 728
 - alibi.models.tensorflow*, 730
 - alibi.models.tensorflow.actor_critic*, 730
 - alibi.models.tensorflow.autoencoder*, 731
 - alibi.models.tensorflow.cfml_models*, 733
 - alibi.prototypes*, 735
 - alibi.prototypes.protoselect*, 737
 - alibi.saving*, 779
 - alibi.tests*, 741
 - alibi.tests.utils*, 741
 - alibi.utils*, 742
 - alibi.utils.approximation_methods*, 751
 - alibi.utils.data*, 752
 - alibi.utils.discretizer*, 752
 - alibi.utils.distance*, 753
 - alibi.utils.distributed*, 756
 - alibi.utils.distributions*, 764
 - alibi.utils.download*, 765
 - alibi.utils.frameworks*, 765
 - alibi.utils.gradients*, 765
 - alibi.utils.kernel*, 766

alibi.utils.lang_model, 767
 alibi.utils.mapping, 772
 alibi.utils.missing_optional_dependency, 774
 alibi.utils.tf, 775
 alibi.utils.visualization, 775
 alibi.utils.wrappers, 778
 alibi.version, 780
 multidim_scaling() (in module alibi.utils.distance), 754
 mvdm() (in module alibi.utils.distance), 755
N
 names() (alibi.explainers.anchors.anchor_explanation.AnchorExplanation method), 624
 negative (alibi.utils.visualization.VisualizeSign attribute), 775
 Neighbors (class in alibi.explainers.anchors.text_samplers), 647
 neighbors() (alibi.explainers.anchors.text_samplers.Neighbors method), 647
 NON_SERIALIZABLE (alibi.explainers.integrated_gradients.LayerState attribute), 701
 NormalActionNoise (class in alibi.explainers.cfrl_base), 693
 not_raises() (in module alibi.tests.utils), 741
 NotFittedError, 779
 num_grad_batch() (in module alibi.utils.gradients), 765
 num_to_ord() (in module alibi.utils.mapping), 772
 NumpyEncoder (class in alibi.saving), 779
O
 ohe_to_ord() (in module alibi.utils), 748
 ohe_to_ord() (in module alibi.utils.mapping), 772
 ohe_to_ord_shape() (in module alibi.utils.mapping), 773
 on_epoch_end() (alibi.explainers.backends.tensorflow.cfrl_base.CFRLBase method), 657
 one_hot_grad() (in module alibi.utils.tf), 775
 ord_to_num() (in module alibi.utils.mapping), 773
 ord_to_ohe() (in module alibi.utils), 749
 ord_to_ohe() (in module alibi.utils.mapping), 773
 order_result() (in module alibi.utils.distributed), 764
 original_image (alibi.utils.visualization.ImageVisualizationMethod attribute), 775
 overlay_mask() (alibi.explainers.AnchorImage method), 580
 overlay_mask() (alibi.explainers.anchors.anchor_image.AnchorImage method), 627
P
 PartialDependence (class in alibi.explainers), 606
 PartialDependence (class in alibi.explainers.partial_dependence), 702
 PartialDependenceBase (class in alibi.explainers.partial_dependence), 703
 PartialDependenceVariance (class in alibi.explainers), 608
 PartialDependenceVariance (class in alibi.explainers.pd_variance), 708
 PermutationImportance (class in alibi.explainers), 610
 PermutationImportance (class in alibi.explainers.permutation_importance), 712
 perturb() (alibi.explainers.CEM method), 589
 perturb() (alibi.explainers.cem.CEM method), 684
 perturb() (in module alibi.utils.gradients), 765
 perturb_sentence() (alibi.explainers.anchors.language_model_text_sampler.LanguageModelTextSampler method), 646
 perturb_sentence_similarity() (alibi.explainers.anchors.text_samplers.SimilaritySampler method), 648
 perturbation (alibi.explainers.anchors.anchor_text.AnchorText attribute), 642
 perturbation (alibi.explainers.AnchorText attribute), 586
 perturbation() (alibi.explainers.anchors.anchor_image.AnchorImage method), 629
 perturbation() (alibi.explainers.anchors.anchor_tabular.TabularSample method), 635
 plot_ale() (in module alibi.explainers), 616
 plot_ale() (in module alibi.explainers.ale), 682
 plot_pd() (in module alibi.explainers), 616
 plot_pd() (in module alibi.explainers.partial_dependence), 707
 plot_pd_variance() (in module alibi.explainers), 617
 plot_pd_variance() (in module alibi.explainers.pd_variance), 710
 plot_permutation_importance() (in module alibi.explainers), 618
 plot_permutation_importance() (in module alibi.explainers.permutation_importance), 714
 PoolCollection (class in alibi.utils.distributed), 761
 positive (alibi.utils.visualization.VisualizeSign attribute), 775
 Postprocessing (class in alibi.explainers.cfrl_base), 693
 precision() (alibi.explainers.anchors.anchor_explanation.AnchorExplanation method), 625
 predict() (alibi.tests.utils.MockPredictor method), 741
 predict_batch_lm() (alibi.utils.lang_model.LanguageModel method), 770
 predict_batch_lm() (alibi.utils.LanguageModel method), 770

method), 747

`predict_batches()` (alibi.explainers.backends.cfml_base.CounterfactualRLDataset method), 627

static method), 664

`predictor` (alibi.explainers.anchors.anchor_tabular.AnchorTabular property), 632

`predictor` (alibi.explainers.AnchorTabular property), 583

`Predictor` (class in alibi.utils.wrappers), 778

`PredictorCallError`, 779

`PredictorReturnTypeError`, 779

`propose_anchors()` (alibi.explainers.anchors.anchor_base.AnchorBaseBase method), 622

`ProtoSelect` (class in alibi.prototypes), 735

`ProtoSelect` (class in alibi.prototypes.prototype), 737

`PtCounterfactualRLDataset` (class in alibi.explainers.backends.pytorch.cfml_base), 650

`PYTORCH` (alibi.utils.frameworks.Framework attribute), 765

R

`R_tilde` (alibi.explainers.cfml_base.ReplayBuffer attribute), 694

`rank_by_importance()` (in module alibi.explainers.shap_wrappers), 721

`RATIO` (alibi.explainers.permutation_importance.Kind attribute), 711

`Reduction` (class in alibi.models.pytorch.metrics), 728

`REGRESSION` (alibi.explainers.similarity.grad.Task attribute), 677

`remote_explainer_index` (alibi.utils.distributed.PoolCollection property), 762

`RemoteSampler` (class in alibi.explainers.anchors.anchor_tabular_distributed), 639

`replace_features()` (alibi.explainers.anchors.anchor_tabular.TabularSampler method), 636

`ReplayBuffer` (class in alibi.explainers.cfml_base), 694

`reset()` (alibi.models.pytorch.metrics.LossContainer method), 727

`reset()` (alibi.models.pytorch.metrics.Metric method), 728

`reset_predictor()` (alibi.api.interfaces.Explainer method), 564

`reset_predictor()` (alibi.explainers.ALE method), 578

`reset_predictor()` (alibi.explainers.ale.ALE method), 679

`reset_predictor()` (alibi.explainers.AnchorImage method), 580

`reset_predictor()` (alibi.explainers.anchors.anchor_image.AnchorImage method), 627

`reset_predictor()` (alibi.explainers.anchors.anchor_tabular.AnchorTabular method), 632

`reset_predictor()` (alibi.explainers.anchors.anchor_tabular_distributed.DistributedAnchorTabular method), 638

`reset_predictor()` (alibi.explainers.anchors.anchor_text.AnchorText method), 642

`reset_predictor()` (alibi.explainers.AnchorTabular method), 583

`reset_predictor()` (alibi.explainers.AnchorText method), 586

`reset_predictor()` (alibi.explainers.CEM method), 589

`reset_predictor()` (alibi.explainers.cem.CEM method), 685

`reset_predictor()` (alibi.explainers.cfproto.CounterfactualProto method), 688

`reset_predictor()` (alibi.explainers.cfml_base.CounterfactualRL method), 691

`reset_predictor()` (alibi.explainers.Counterfactual method), 591

`reset_predictor()` (alibi.explainers.counterfactual.Counterfactual method), 699

`reset_predictor()` (alibi.explainers.CounterfactualProto method), 595

`reset_predictor()` (alibi.explainers.CounterfactualRL method), 596

`reset_predictor()` (alibi.explainers.DistributedAnchorTabular method), 600

`reset_predictor()` (alibi.explainers.integrated_gradients.IntegratedGradients method), 701

`reset_predictor()` (alibi.explainers.IntegratedGradients method), 603

`reset_predictor()` (alibi.explainers.KernelShap method), 606

`reset_predictor()` (alibi.explainers.partial_dependence.PartialDependenceBase method), 705

`reset_predictor()` (alibi.explainers.permutation_importance.PermutationImportance method), 713

- [reset_predictor\(\)](#) (*alibi.explainers.PermutationImportance* method), [611](#)
[reset_predictor\(\)](#) (*alibi.explainers.shap_wrappers.KernelShap* method), [718](#)
[reset_predictor\(\)](#) (*alibi.explainers.shap_wrappers.TreeShap* method), [720](#)
[reset_predictor\(\)](#) (*alibi.explainers.similarity.base.BaseSimilarityExplainer* method), [674](#)
[reset_predictor\(\)](#) (*alibi.explainers.TreeShap* method), [616](#)
[ResourceError](#), [762](#)
[result\(\)](#) (*alibi.models.pytorch.metrics.LossContainer* method), [727](#)
[result\(\)](#) (*alibi.models.pytorch.metrics.Metric* method), [728](#)
[return_attribute\(\)](#) (*alibi.explainers.shap_wrappers.KernelExplainerWrapper* method), [715](#)
[return_attribute\(\)](#) (*alibi.utils.distributed.DistributedExplainer* method), [761](#)
[return_attribute\(\)](#) (*alibi.utils.DistributedExplainer* method), [745](#)
[Riemann](#) (class in *alibi.utils.approximation_methods*), [751](#)
[riemann_builders\(\)](#) (in module *alibi.utils.approximation_methods*), [751](#)
[right](#) (*alibi.utils.approximation_methods.Riemann* attribute), [751](#)
[RobertaBase](#) (class in *alibi.utils*), [748](#)
[RobertaBase](#) (class in *alibi.utils.lang_model*), [771](#)
[round_grad\(\)](#) (in module *alibi.utils.tf*), [775](#)
- ## S
- [sample\(\)](#) (*alibi.explainers.cfml_base.ReplayBuffer* method), [695](#)
[sample\(\)](#) (in module *alibi.explainers.backends.cfml_tabular*), [671](#)
[sample_categorical\(\)](#) (in module *alibi.explainers.backends.cfml_tabular*), [672](#)
[sample_differentiable\(\)](#) (in module *alibi.explainers.backends.pytorch.cfml_tabular*), [656](#)
[sample_differentiable\(\)](#) (in module *alibi.explainers.backends.tensorflow.cfml_tabular*), [663](#)
[sample_numerical\(\)](#) (in module *alibi.explainers.backends.cfml_tabular*), [672](#)
[sampler\(\)](#) (*alibi.explainers.anchors.anchor_text.AnchorText* method), [643](#)
[sampler\(\)](#) (*alibi.explainers.AnchorText* method), [586](#)
[samplers](#) (*alibi.explainers.anchors.anchor_tabular.AnchorTabular* attribute), [632](#)
[SampleTabularPostprocessing](#) (class in *alibi.explainers.cfml_tabular*), [697](#)
[SAMPLING_LANGUAGE_MODEL](#) (*alibi.explainers.anchors.anchor_text.AnchorText* attribute), [640](#)
[SAMPLING_LANGUAGE_MODEL](#) (*alibi.explainers.AnchorText* attribute), [583](#)
[SAMPLING_SIMILARITY](#) (*alibi.explainers.anchors.anchor_text.AnchorText* attribute), [640](#)
[SAMPLING_SIMILARITY](#) (*alibi.explainers.AnchorText* attribute), [583](#)
[SAMPLING_UNKNOWN](#) (*alibi.explainers.anchors.anchor_text.AnchorText* attribute), [640](#)
[SAMPLING_UNKNOWN](#) (*alibi.explainers.AnchorText* attribute), [583](#)
[save\(\)](#) (*alibi.api.interfaces.Explainer* method), [564](#)
[save\(\)](#) (*alibi.api.interfaces.Summariser* method), [565](#)
[save\(\)](#) (*alibi.explainers.cfml_base.CounterfactualRL* method), [691](#)
[save\(\)](#) (*alibi.explainers.CounterfactualRL* method), [597](#)
[save_explainer\(\)](#) (in module *alibi.saving*), [780](#)
[save_model\(\)](#) (in module *alibi.explainers.backends.pytorch.cfml_base*), [653](#)
[save_model\(\)](#) (in module *alibi.explainers.backends.tensorflow.cfml_base*), [661](#)
[save_weights\(\)](#) (*alibi.models.pytorch.model.Model* method), [730](#)
[scale_image\(\)](#) (in module *alibi.explainers.anchors.anchor_image*), [629](#)
[score\(\)](#) (*alibi.confidence.LinearityMeasure* method), [566](#)
[score\(\)](#) (*alibi.confidence.model_linearity.LinearityMeasure* method), [569](#)
[score\(\)](#) (*alibi.confidence.TrustScore* method), [568](#)
[score\(\)](#) (*alibi.confidence.trustscore.TrustScore* method), [571](#)
[score\(\)](#) (*alibi.explainers.cfproto.CounterfactualProto* method), [688](#)
[score\(\)](#) (*alibi.explainers.CounterfactualProto* method), [595](#)
[SCORE_FNS](#) (in module *alibi.explainers.permutation_importance*), [713](#)
[seed\(\)](#) (*alibi.explainers.anchors.language_model_text_sampler.LanguageModelTextSampler* method), [646](#)
[select_critical_arms\(\)](#) (*alibi.explainers.anchors.anchor_base.AnchorBaseBeam* method), [646](#)

method), 622

select_word() (alibi.utils.lang_model.LanguageModel method), 771

select_word() (alibi.utils.LanguageModel method), 747

SerializationError, 779

set_actor_index() (alibi.utils.distributed.DistributedExplainer method), 761

set_actor_index() (alibi.utils.DistributedExplainer method), 745

set_data_type() (alibi.explainers.anchors.language_model_text_sampler.LanguageModelTextSampler method), 646

set_data_type() (alibi.explainers.anchors.text_samplers.SimilaritySampler method), 649

set_data_type() (alibi.explainers.anchors.text_samplers.UnknownSampler method), 650

set_instance_label() (alibi.explainers.anchors.anchor_tabular.TabularSampler method), 636

set_instance_label() (alibi.explainers.anchors.anchor_tabular_distributed.TabularSampler method), 639

set_n_covered() (alibi.explainers.anchors.anchor_tabular.TabularSampler method), 636

set_n_covered() (alibi.explainers.anchors.anchor_tabular_distributed.RemoteSampler method), 639

set_seed() (in module alibi.explainers.backends.pytorch.cfrl_base), 654

set_seed() (in module alibi.explainers.backends.tensorflow.cfrl_base), 661

set_text() (alibi.explainers.anchors.language_model_text_sampler.LanguageModelTextSampler method), 647

set_text() (alibi.explainers.anchors.text_samplers.AnchorTextSampler method), 647

set_text() (alibi.explainers.anchors.text_samplers.SimilaritySampler method), 649

set_text() (alibi.explainers.anchors.text_samplers.UnknownSampler method), 650

sigma (alibi.utils.kernel.GaussianRBF property), 766

SimilaritySampler (class in alibi.explainers.anchors.text_samplers), 647

spacy_model() (in module alibi.utils), 749

spacy_model() (in module alibi.utils.download), 765

sparsity_loss() (in module alibi.explainers.backends.pytorch.cfrl_base), 654

sparsity_loss() (in module alibi.explainers.backends.pytorch.cfrl_tabular), 656

sparsity_loss() (in module alibi.explainers.backends.tensorflow.cfrl_base), 661

sparsity_loss() (in module alibi.explainers.backends.tensorflow.cfrl_tabular), 664

split_ohe() (in module alibi.explainers.backends.cfrl_tabular), 672

squared_pairwise_distance() (in module alibi.explainers.backends.cfrl_tabular), 672

submit() (alibi.utils.distributed.ActorPool method), 758

SUBWORD_PREFIX (alibi.utils.BertBaseUncased attribute), 742

SUBWORD_PREFIX (alibi.utils.DistilbertBaseUncased attribute), 742

SUBWORD_PREFIX (alibi.utils.lang_model.BertBaseUncased attribute), 767

SUBWORD_PREFIX (alibi.utils.lang_model.DistilbertBaseUncased attribute), 768

SUBWORD_PREFIX (alibi.utils.lang_model.LanguageModel attribute), 769

SUBWORD_PREFIX (alibi.utils.lang_model.RobertaBase attribute), 771

SUBWORD_PREFIX (alibi.utils.LanguageModel attribute), 745

SUBWORD_PREFIX (alibi.utils.RobertaBase attribute), 748

SUM (alibi.models.pytorch.metrics.Reduction attribute), 728

sum_categories() (in module alibi.explainers.shap_wrappers), 721

summarise() (alibi.api.interfaces.Summariser method), 565

summarise() (alibi.prototypes.ProtoSelect method), 736

summarise() (alibi.prototypes.protoselect.ProtoSelect method), 738

Summariser (class in alibi.api.interfaces), 565

SUPPORTED_RIEMANN_METHODS (in module alibi.utils.approximation_methods), 751

TabularSampler (class in alibi.explainers.anchors.anchor_tabular), 632

Task (class in alibi.explainers.similarity.grad), 677

TENSORFLOW (alibi.utils.frameworks.Framework attribute), 765

test_step() (alibi.models.pytorch.model.Model method), 730

TfCounterfactualRLDataset (class in alibi.explainers.backends.tensorflow.cfrl_base), 657

[to_disk\(\)](#) (*alibi.utils.lang_model.LanguageModel* method), 771
[to_disk\(\)](#) (*alibi.utils.LanguageModel* method), 747
[to_json\(\)](#) (*alibi.api.interfaces.Explanation* method), 565
[to_numpy\(\)](#) (in module *alibi.explainers.backends.pytorch.cfrl_base*), 654
[to_numpy\(\)](#) (in module *alibi.explainers.backends.tensorflow.cfrl_base*), 661
[to_sample\(\)](#) (*alibi.explainers.anchors.anchor_base.AnchorBaseBeam* static method), 623
[to_tensor\(\)](#) (in module *alibi.explainers.backends.pytorch.cfrl_base*), 654
[to_tensor\(\)](#) (in module *alibi.explainers.backends.tensorflow.cfrl_base*), 661
[tokenizer](#) (*alibi.utils.lang_model.BertBaseUncased* attribute), 768
[tokenizer](#) (*alibi.utils.lang_model.DistilbertBaseUncased* attribute), 769
[tokenizer](#) (*alibi.utils.lang_model.LanguageModel* attribute), 771
[tokenizer](#) (*alibi.utils.lang_model.RobertaBase* attribute), 772
[tokenizer](#) (*alibi.utils.LanguageModel* attribute), 748
[train_step\(\)](#) (*alibi.models.pytorch.model.Model* method), 730
[trapezoid](#) (*alibi.utils.approximation_methods.Riemann* attribute), 751
[TREE_SHAP_PARAMS](#) (in module *alibi.api.defaults*), 563
[TreePartialDependence](#) (class in *alibi.explainers*), 612
[TreePartialDependence](#) (class in *alibi.explainers.partial_dependence*), 705
[TreeShap](#) (class in *alibi.explainers*), 613
[TreeShap](#) (class in *alibi.explainers.shap_wrappers*), 718
[TrustScore](#) (class in *alibi.confidence*), 566
[TrustScore](#) (class in *alibi.confidence.trustscore*), 570

U

[UNK](#) (*alibi.explainers.anchors.text_samplers.UnknownSampler* attribute), 649
[UnknownSampler](#) (class in *alibi.explainers.anchors.text_samplers*), 649
[UNSPECIFIED](#) (*alibi.explainers.integrated_gradients.LayerState* attribute), 701
[update_actor_critic\(\)](#) (in module *alibi.explainers.backends.pytorch.cfrl_base*), 654
[update_actor_critic\(\)](#) (in module *alibi.explainers.backends.tensorflow.cfrl_base*), 662

V

[update_state\(\)](#) (*alibi.explainers.anchors.anchor_base.AnchorBaseBeam* method), 623
[update_state\(\)](#) (*alibi.models.pytorch.metrics.Metric* method), 728
[validate_prediction_labels\(\)](#) (*alibi.models.pytorch.model.Model* method), 730
[visualize_image_attr\(\)](#) (in module *alibi.utils*), 749
[visualize_image_attr\(\)](#) (in module *alibi.utils.visualization*), 777
[visualize_image_prototypes\(\)](#) (in module *alibi.prototypes*), 736
[visualize_image_prototypes\(\)](#) (in module *alibi.prototypes.protoselect*), 740
[VisualizeSign](#) (class in *alibi.utils.visualization*), 775

X

[X](#) (*alibi.explainers.cfrl_base.ReplayBuffer* attribute), 694

Y

[Y_m](#) (*alibi.explainers.cfrl_base.ReplayBuffer* attribute), 694
[Y_t](#) (*alibi.explainers.cfrl_base.ReplayBuffer* attribute), 694

Z

[Z](#) (*alibi.explainers.cfrl_base.ReplayBuffer* attribute), 694
[Z_cf_tilde](#) (*alibi.explainers.cfrl_base.ReplayBuffer* attribute), 694