alibi Documentation

Release 0.2.1

Seldon Technologies Ltd

Jul 02, 2019

Contents

Alibi is an open source Python library aimed at machine learning model inspection and interpretation. The initial focus on the library is on black-box, instance based model explanations.

- Provide high quality reference implementations of black-box ML model explanation algorithms
- Define a consistent API for interpretable ML methods
- Support multiple use cases (e.g. tabular, text and image data classification, regression)
- Implement the latest model explanation, concept drift, algorithmic bias detection and other ML model monitoring and interpretation methods

CHAPTER 1

Getting Started

1.1 Installation

Alibi works with Python 3.5+ and can be installed from PyPI:

```
pip install alibi
```

1.2 Features

Alibi is a Python package designed to help explain the predictions of machine learning models, gauge the confidence of predictions and eventually support wider capabilities of inspecting the performance of models with respect to concept drift and algorithmic bias. 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__
```

```
['AnchorTabular',
 'AnchorText',
 'AnchorImage',
 'CEM',
 'CounterFactual',
 'CounterFactualProto']
```

For gauging model confidence:

```
alibi.confidence.__all__
```

```
['TrustScore']
```

For detailed information on the methods:

- *Overview of available methods*
  - *Anchor explanations*
  - *Contrastive Explanation Method (CEM)*
  - *Counterfactual Instances*
  - *Counterfactuals Guided by Prototypes*
  - *Trust Scores*

1.3 Basic Usage

We will use the *Anchor method on tabular data* to illustrate the usage of explainers in Alibi.

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)
```

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

```
explainer.explain(x)
```

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.

CHAPTER 2

Algorithm overview

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

2.1 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?" The following table summarizes the capabilities of the current algorithms:

Explainer	Classification	Regression	Categorical features	Tabular	Text	Images	Needs training set
Anchors	✓		✓	✓	✓	✓	For Tabular
CEM	✓			✓		✓	Optional
Counterfactual Instances	✓			✓		✓	No
Prototype Counterfactuals	✓			✓		✓	Optional

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 me 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 instances: generate counterfactual examples using a simple loss function. *Documentation*, *image classification*.

Prototype Counterfactuals: 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*, *image classification*.

2.2 Model Confidence

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

Algorithm	Classification	Regression	Categorical features	Tabular	Text	Images	Needs training set
Trust Scores	✓			✓	✓[1]	✓[2]	Yes

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*

[1] Depending on model
[2] May require dimensionality reduction

CHAPTER 3

Roadmap

Alibi aims to be the go-to library for ML model interpretability and monitoring. There are multiple challenges for developing a high-quality, production-ready library that achieves this. In addition to having high quality reference implementations of the most promising algorithms, we need extensive documentation and case studies comparing the different interpretability methods and their respective pros and cons. A clean and a usable API is also a priority. Additionally we want to move beyond model explanation and provide tools to gauge ML model confidence, measure concept drift, detect outliers and algorithmic bias among other things.

3.1 Additional explanation methods

- Influence functions [WIP]
- Feature attribution methods (e.g. SHAP)
- Global methods (e.g. ALE)

3.2 Important enhancements to explanation methods

- Robust handling of categorical variables (Github issue)
- Document pitfalls of popular methods like LIME and PDP (Github issue)
- Unified API (Github issue)
- Standardized return types for explanations
- Explanations for regression models (Github issue)
- Explanations for sequential data
- Develop methods for highly correlated features

3.3 Beyond explanations

- Investigate alternatives to Trust Scores for gauging the confidence of black-box models
- Concept drift - provide methods for monitoring and alerting to changes in the incoming data distribution and the conditional distribution of the predictions
- Bias detection methods
- Outlier detection methods (Github issue)

[source]

CHAPTER 4

Anchors

4.1 Overview

The anchor algorithm is based on the Anchors: High-Precision Model-Agnostic Explanations paper by Ribeiro et al. 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.

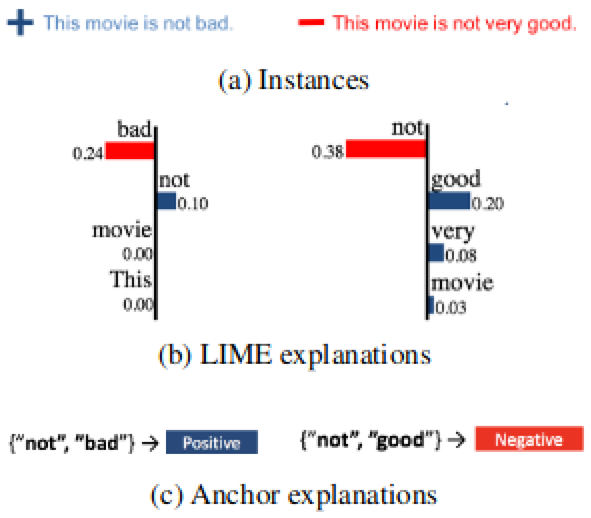


Figure 1: Sentiment predictions, LSTM

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.

4.1.1 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 2 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.

4.1.2 Tabular Data

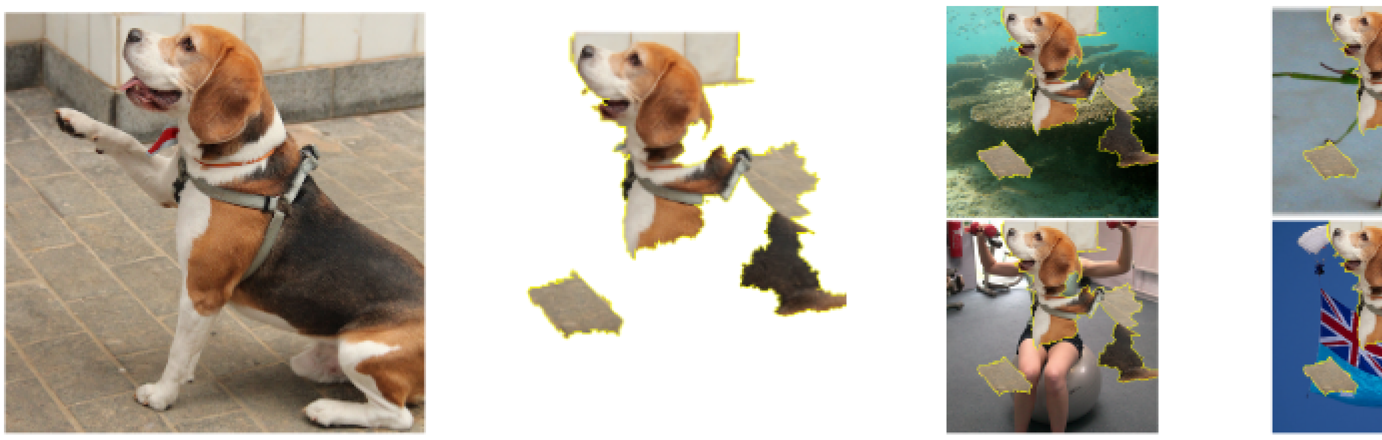
Anchors are also suitable for tabular data with both categorical and continuous features. The continuous features are discretized into quantiles (e.g. deciles), so they become more interpretable. The features in a candidate anchor are kept constant (same category or bin for discretized features) while 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.

4.1.3 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(be

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

4.1.4 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.

4.2 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`: the previously discussed minimal confidence level. `threshold` defines the minimum fraction of samples for a candidate anchor that need to lead to the same prediction as the original instance. A higher value gives more confidence in the anchor, but also leads to more computation time. The default value is 0.95.
- `tau`: determines when we assume convergence for the multi-armed bandit. A bigger value for `tau` means faster convergence but also looser anchor conditions. By default equal to 0.15.
- `beam_size`: the size of the beam width. A bigger beam width can lead to a better overall anchor at the expense of more computation time.
- `batch_size`: the batch size used for sampling. A bigger 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 compute the coverage of the anchor. By default set to 10000.

4.2.1 Text

Initialization

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))
```

If we choose to sample similar words from a corpus, we first need to load a spaCy model:

```
import spacy
from alibi.utils.download import spacy_model

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

We can now initialize our explainer:

```
explainer = AnchorText(nlp, predict_fn)
```

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, use_proba=True, use_unk=False)
```

We set the confidence `threshold` at 95%. `use_proba` equals True means that we will sample from the corpus proportional to the word similarity, while `use_unk` False implies that we are not replacing words outside the anchor with UNK tokens. `use_proba` False and `use_unk` True would mean that we are simply replacing the words that are not in the candidate anchor with the UNK tokens.

The `explain` method returns a dictionary that contains *key: value* pairs for:

- *names*: the 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 coverage of the anchor over a sampled part of the training set.

Under the *raw* key, the dictionary 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 *names*, *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.

4.2.2 Tabular Data

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: list('married', 'divorced'), 3: list('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.

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

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])
```

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 dictionary contains the same *key: value* pairs 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
```

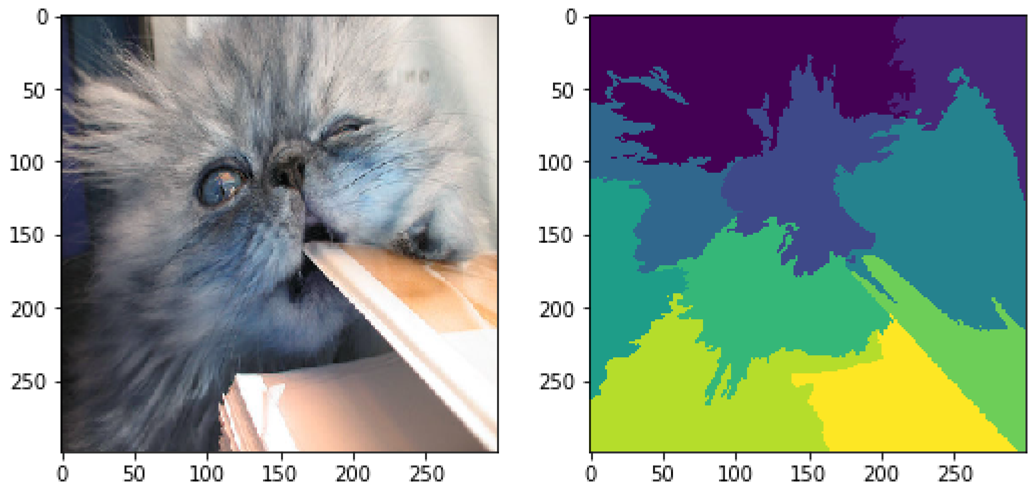
4.2.3 Images

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.

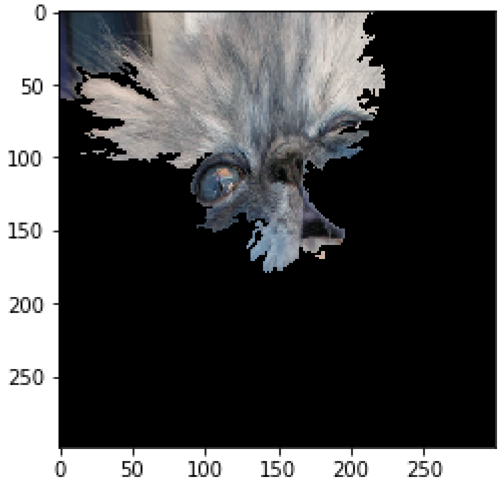
Explanation

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 dictionary 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* key (see image below) as well as the image's superpixels under *segments*.



4.3 Examples

4.3.1 Image

Anchor explanations for ImageNet

Anchor explanations for fashion MNIST

4.3.2 Tabular Data

Anchor explanations on the Iris dataset

Anchor explanations for income prediction

4.3.3 Text

Anchor explanations for movie sentiment

[source]

CHAPTER 5

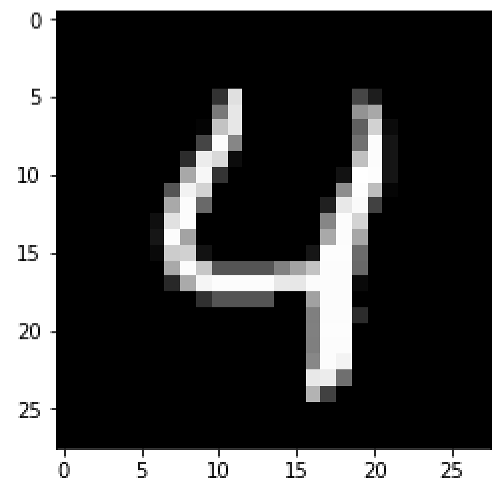
Contrastive Explanation Method

5.1 Overview

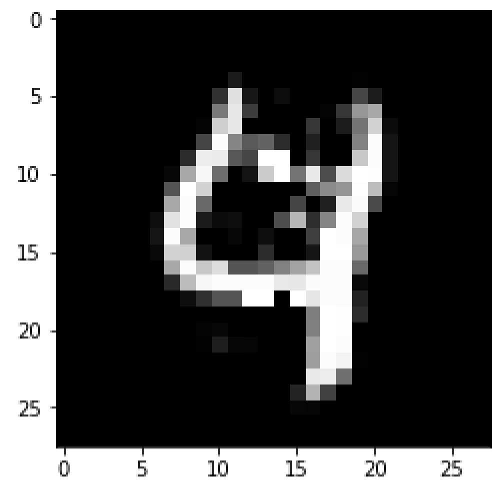
The *Contrastive Explanation Method* (CEM) is based on the paper Explanations based on the Missing: Towards Constrastive Explanations with Pertinent Negatives and extends the code open sourced by the authors. CEM generates instance based local black box explanations for classification models in terms 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, ..., f_k are present and because features f_m, ..., 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.

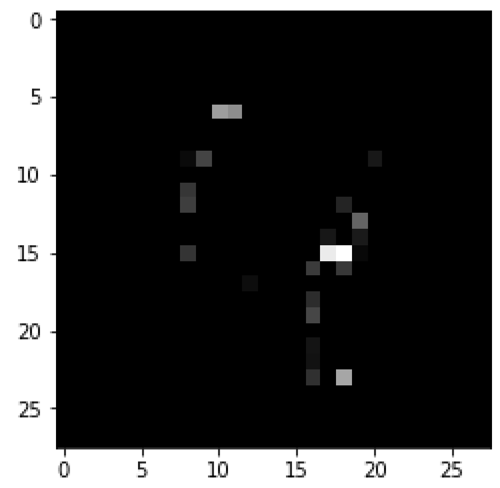
The ability to add or remove features to arrive at respectively PN's or PP's implies that there are feature values that contain no information with regards to the model's predictions. Consider for instance the MNIST image below where the pixels are scaled between 0 and 1. The pixels with values close 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.

5.2 Usage

5.2.1 Initialization

Because the optimizer is defined in TensorFlow (TF), we need to run the CEM explainer within a TensorFlow session:

```
# initialize TensorFlow session before model definition
sess = tf.Session()
K.set_session(sess)  # using a Keras model in the same session
sess.run(tf.global_variables_initializer())
```

We can then 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(sess, 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 previously defined session as well as 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.

5.2.2 Explanation

We can finally explain the instance and close the TensorFlow session when we are done:

```
explanation = cem.explain(X)
sess.close()
K.clear_session()
```

The `explain` method returns a dictionary with the following *key: value* pairs:

- *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

5.2.3 Numerical Gradients

So far, the whole optimization problem could be defined within the 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. The CEM can be initialized in the TF session as follows:

```
# 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(sess, 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 x nb of prediction categories)* and for `eps[1]` it should be *(1 x 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.

5.3 Examples

Contrastive Explanations Method (CEM) applied to MNIST

Contrastive Explanations Method (CEM) applied to Iris dataset

[source]

CHAPTER 6

Counterfactual Instances

6.1 Overview

A counterfactual explanation of an outcome or a situation Y takes the form "If X had not occured, Y would not have occured" (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. There main issue with this definition of "interpretability" is that the difference between X' and X required to change the model prediciton 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 lost 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. $\lambda \in (10^{-1}, \ldots, 10^{-10})$ to find lower and upper bounds $\lambda_{\text{lb}}, \lambda_{\text{ub}}$ where counterfactuals exist. Then we use bisection to find the maximum $\lambda \in [\lambda_{\text{lb}}, \lambda_{\text{ub}}]$ such that the counterfactual constraint still holds. The result is a set of counterfactual instances X' with varying distance from the test instance X.

6.2 Usage

6.2.1 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*.

Similar to other methods, we use TensorFlow (TF) internally to solve the optimization problem defined above, thus we need to run the counterfactual explainer within a TF session, for a Keras model once it has been loaded we can just get it:

```
model = load_model('my_model.h5')
sess = K.get_session()
```

Then we can initialize the counterfactual object:

```
shape = (1,) + x_train.shape[1:]
cf = CounterFactual(sess, 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 session and 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.

6.2.2 Fit

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

6.2.3 Explanation

We can now explain the instance X and close the TensorFlow session when we are done:

```
explanation = cf.explain(X)
sess.close()
K.clear_session()
```

The `explain` method returns a dictionary with the following *key: value* pairs:

- *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 probabilites 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.

6.2.4 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 TF session as follows:

```python
# 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(sess, 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 x nb of features)*, then a separate perturbation value can be used for each feature. For the Iris dataset, `eps` could look as follows:

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

6.3 Examples

Counterfactual instances on MNIST

[source]

CHAPTER 7

Counterfactuals Guided by Prototypes

7.1 Overview

According to Molnar's Interpretable Machine Learning book, a counterfactual explanation of a prediction describes the smallest change to the feature values that changes the prediction. One use case for counterfactuals would be loan approvals. Ann might be interested why her application for a loan was rejected, and what would need to change so the application is approved.

Counterfactuals are generated from the original instance by applying perturbations. The counterfactual then needs to satisfy certain constraints related to the model prediction or sparsity of the perturbed instance.

Some issues arise during the perturbation process:

- the training data manifold needs to be respected
- finding a satisfactory counterfactual can take time, especially for high dimensional data
- there is often a trade off between sparsity and interpretability of the counterfactual

We can address these issues 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 $\beta L_1 + L_2$ acts as a regularizer and introduces 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 manifold is not taken into account, and the perturbations are not necessarily meaningful.

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

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

The L_{AE} does however not necessarily lead to interpretable solutions or speed up the counterfactual search. 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 auto-encoder. 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 encoding for that class. 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 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:

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

The method allows us to select specific prototype classes to guide the counterfactual. For example, in MNIST the closest prototype to 9 is 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 predict 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 predict 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 28x28x2 = 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.

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

More details will be revealed in a forthcoming paper.

7.2 Usage

7.2.1 Initialization

The counterfactuals guided by prototypes 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. Please see the *numerical gradients* section for black box models.

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

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

Because the optimizer is defined in TensorFlow, we need to run the explainer within a TensorFlow session. The example below uses the session set by the loaded Keras models:

```
sess = K.get_session()
```

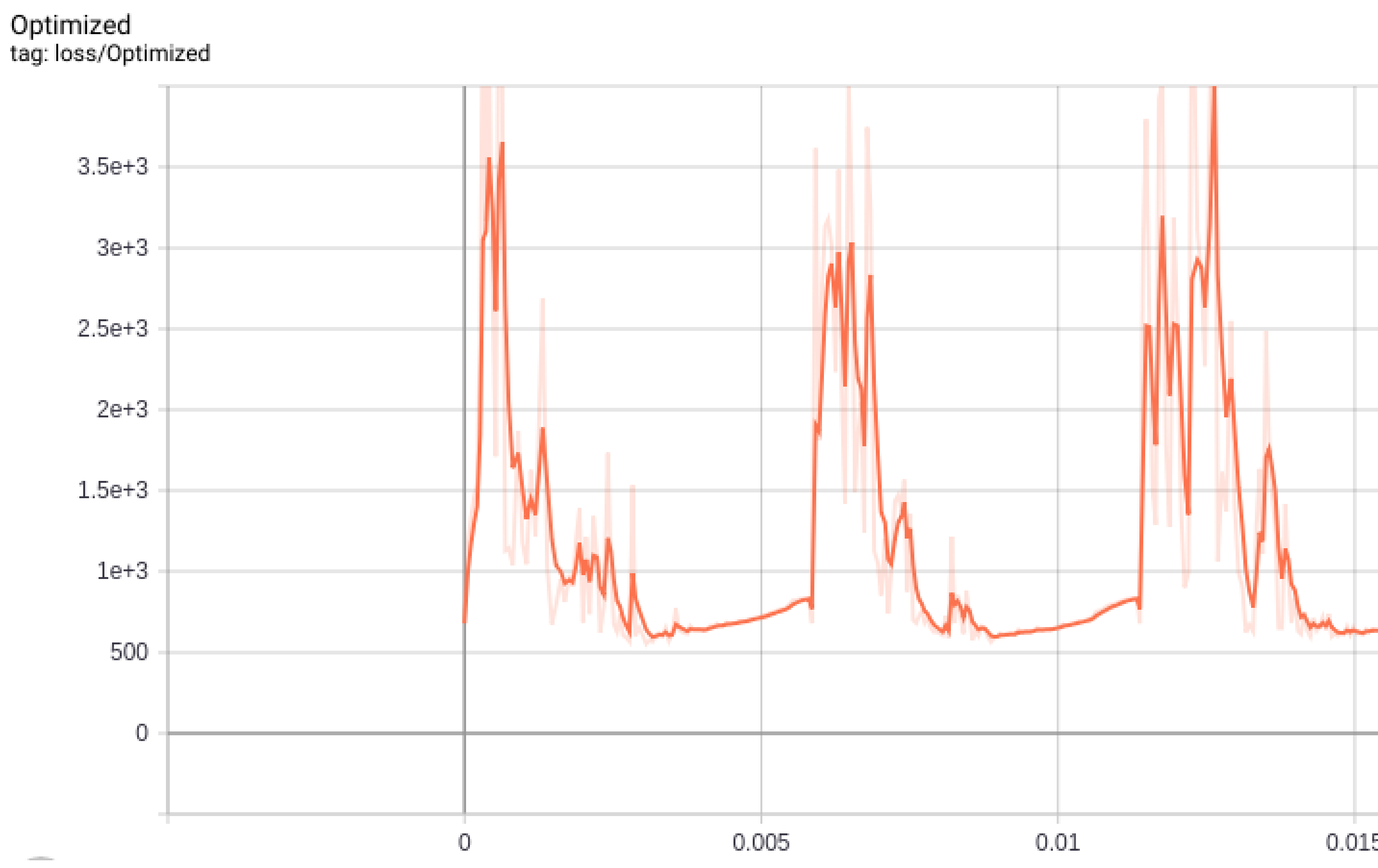
We can now initialize the counterfactual:

```
shape = (1,) + x_train.shape[1:]
cf = CounterFactualProto(sess, 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 previously defined session as well as the predictive, and (optional) auto-encoder 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 auto-encoder. 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.

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.

7.2.2 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)
```

7.2.3 Explanation

We can now explain the instance and close the TensorFlow session when we are done:

```
explanation = cf.explain(X, Y=None, target_class=None, threshold=0.,
                         verbose=True, print_every=100, log_every=100)
sess.close()
K.clear_session()
```

- `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*.
- `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 a dictionary with the following *key: value* pairs:

- *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.

7.2.4 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 TF session as follows:

```
# 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')

sess = K.get_session()

# initialize explainer
shape = (1,) + x_train.shape[1:]
cf = CounterFactualProto(sess, 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 predict function loss term, 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 ${}^{\delta L_{pred}}/_{\delta p}$ 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 x nb of prediction categories)* and for `eps[1]` it should be *(1 x 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.

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(sess, 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)
```

7.2.5 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 closest class instance as the nearest prototype. We can run the counterfactual as follows:

```
cf = CounterFactualProto(sess, cnn, shape, use_kdtree=True, theta=10., feature_
↪range=(-.5, .5))
cf.fit(x_train, trustscore_kwargs=None)
explanation = cf.explain(X)
```

- `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.

7.3 Examples

Counterfactuals guided by prototypes on MNIST

Counterfactuals guided by prototypes on Boston housing dataset

[source]

CHAPTER 8

Trust Scores

8.1 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 kth nearest neighbor in each tree or by using the average distance from the first to the kth 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*.

8.2 Usage

8.2.1 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 kth 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.

8.2.2 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`: kth 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 kth nearest point (*mean*).

8.3 Examples

Trust Scores applied to Iris

Trust Scores applied to MNIST

CHAPTER 9

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 adult
```

9.1 Load adult dataset

[2]:
```
data, labels, feature_names, category_map = adult()
```

```
/home/avl/git/fork-alibi/alibi/datasets.py:126: ParserWarning: Falling back to the␣
↪'python' engine because the 'c' engine does not support regex separators␣
↪(separators > 1 char and different from '\s+' are interpreted as regex); you can␣
↪avoid this warning by specifying engine='python'.
  raw_data = pd.read_csv(dataset_url, names=raw_features, delimiter=', ').fillna('?')
```

Define shuffled training and test set

[3]:
```
np.random.seed(0)
data_perm = np.random.permutation(np.c_[data, labels])
data = data_perm[:,:-1]
labels = data_perm[:,-1]
```

```
[4]: idx = 30000
     X_train,Y_train = data[:idx,:], labels[:idx]
     X_test, Y_test = data[idx+1:,:], labels[idx+1:]
```

9.2 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:

```
[5]: 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:

```
[6]: categorical_features = list(category_map.keys())
     categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median
     ↪')),
                                               ('onehot', OneHotEncoder(handle_unknown=
     ↪'ignore'))])
```

Combine and fit:

```
[7]: preprocessor = ColumnTransformer(transformers=[('num', ordinal_transformer, ordinal_
     ↪features),
                                                    ('cat', categorical_transformer,␣
     ↪categorical_features)])
     preprocessor.fit(data)
```

```
[7]: ColumnTransformer(n_jobs=None, remainder='drop', sparse_threshold=0.3,
              transformer_weights=None,
              transformers=[('num', Pipeline(memory=None,
          steps=[('imputer', SimpleImputer(copy=True, fill_value=None, missing_values=nan,
            strategy='median', verbose=0)), ('scaler', StandardScaler(copy=True, with_
     ↪mean=True, with_std=True))]), [0, 8, 9, 10]), ('cat', Pipeline(memory=None,
          steps=[(...oat64'>, handle_unknown='ignore',
            n_values=None, sparse=True))]), [1, 2, 3, 4, 5, 6, 7, 11])])
```

9.3 Train Random Forest model

Fit on pre-processed (imputing, OHE, standardizing) data.

```
[8]: np.random.seed(0)
     clf = RandomForestClassifier(n_estimators=50)
     clf.fit(preprocessor.transform(X_train), Y_train)
```

```
[8]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                 max_depth=None, max_features='auto', max_leaf_nodes=None,
                 min_impurity_decrease=0.0, min_impurity_split=None,
```

(continues on next page)

(continued from previous page)

```
            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

```
[9]: 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:  0.85390625
```

9.4 Initialize and fit anchor explainer for tabular data

```
[10]: explainer = AnchorTabular(predict_fn, feature_names, categorical_names=category_map)
```

Discretize the ordinal features into quartiles

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

9.5 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.

```
[12]: idx = 0
      class_names = ['<=50K', '>50K']
      print('Prediction: ', class_names[explainer.predict_fn(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.

```
[13]: explanation = explainer.explain(X_test[idx], threshold=0.95)
      print('Anchor: %s' % (' AND '.join(explanation['names'])))
      print('Precision: %.2f' % explanation['precision'])
      print('Coverage: %.2f' % explanation['coverage'])
```

```
Anchor: Marital Status = Separated AND Sex = Female
Precision: 0.96
Coverage: 0.11
```

CHAPTER 10

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
```

10.1 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:]
```

10.2 Train Random Forest model

```
[4]: np.random.seed(0)
     clf = RandomForestClassifier(n_estimators=50)
     clf.fit(X_train, Y_train)
```

```
[4]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                 max_depth=None, max_features='auto', max_leaf_nodes=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,
```

(continues on next page)

(continued from previous page)

```
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

Define predict function

```
[5]: predict_fn = lambda x: clf.predict_proba(x)
```

10.3 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])
```

10.4 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.predict_fn(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['names'])))
     print('Precision: %.2f' % explanation['precision'])
     print('Coverage: %.2f' % explanation['coverage'])
```

```
Anchor: petal width (cm) > 1.80 AND sepal width (cm) <= 2.80
Precision: 0.99
Coverage: 0.07
```

CHAPTER 11

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.

```
[1]: 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
     import spacy
     from alibi.explainers import AnchorText
     from alibi.datasets import movie_sentiment
     from alibi.utils.download import spacy_model
```

11.1 Load movie review dataset

```
[2]: data, labels = movie_sentiment()
```

Define shuffled training, validation and test set

```
[3]: 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)
```

11.2 Apply CountVectorizer to training set

```
[4]: vectorizer = CountVectorizer(min_df=1)
     vectorizer.fit(train)
```

```
[4]: CountVectorizer(analyzer='word', binary=False, decode_error='strict',
             dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
             lowercase=True, max_df=1.0, max_features=None, min_df=1,
             ngram_range=(1, 1), preprocessor=None, stop_words=None,
             strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
             tokenizer=None, vocabulary=None)
```

11.3 Fit model

```
[5]: np.random.seed(0)
     clf = LogisticRegression(solver='liblinear')
     clf.fit(vectorizer.transform(train), train_labels)
```

```
[5]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
               intercept_scaling=1, max_iter=100, multi_class='warn',
               n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
               tol=0.0001, verbose=0, warm_start=False)
```

11.4 Define prediction function

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

11.5 Make predictions on train and test sets

```
[7]: preds_train = predict_fn(train)
     preds_val = predict_fn(val)
     preds_test = predict_fn(test)
     print('Train accuracy', accuracy_score(train_labels, preds_train))
     print('Validation accuracy', accuracy_score(val_labels, preds_val))
     print('Test accuracy', accuracy_score(test_labels, preds_test))
```

```
Train accuracy 0.9801624284382905
Validation accuracy 0.7544910179640718
Test accuracy 0.7589841878294202
```

11.6 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.

```
[8]: model = 'en_core_web_md'
     spacy_model(model=model)
     nlp = spacy.load(model)
```

11.7 Initialize anchor text explainer

[9]:
```
explainer = AnchorText(nlp, predict_fn)
```

11.8 Explain a prediction

[10]:
```
class_names = ['negative', 'positive']
```

Prediction:

[11]:
```
text = 'This is a good book .'
pred = class_names[predict_fn([text])[0]]
alternative =  class_names[1 - predict_fn([text])[0]]
print('Prediction: %s' % pred)
```

```
Prediction: positive
```

Explanation:

[12]:
```
np.random.seed(0)
explanation = explainer.explain(text, threshold=0.95, use_proba=False, use_unk=True)
```

use_unk=True means we will perturb examples by replacing words with UNKs. Let us now take a look at the anchor. The word 'good' basically guarantees a positive prediction. This is because the UNKs do not take instances like 'not good' into account.

[13]:
```
print('Anchor: %s' % (' AND '.join(explanation['names'])))
print('Precision: %.2f' % explanation['precision'])
print('\nExamples where anchor applies and model predicts %s:' % pred)
print('\n'.join([x[0] for x in explanation['raw']['examples'][-1]['covered_true']]))
print('\nExamples where anchor applies and model predicts %s:' % alternative)
print('\n'.join([x[0] for x in explanation['raw']['examples'][-1]['covered_false']]))
```

```
Anchor: good
Precision: 1.00

Examples where anchor applies and model predicts positive:
UNK UNK UNK good book UNK
UNK is a good book .
UNK is a good book UNK
UNK is UNK good book .
UNK UNK UNK good book .
UNK is a good book .
UNK is UNK good UNK UNK
UNK UNK UNK good UNK .
This UNK a good UNK UNK
This is a good UNK .

Examples where anchor applies and model predicts negative:
```

11.9 Changing the perturbation distribution

Let's try this with another perturbation distribution, namely one that replaces words by similar words instead of UNKs.

Explanation:

```
[14]: np.random.seed(0)
explanation = explainer.explain(text, threshold=0.95, use_proba=True, use_unk=False)
```

The anchor now shows that we need more to guarantee the positive prediction:

```
[15]: print('Anchor: %s' % (' AND '.join(explanation['names'])))
print('Precision: %.2f' % explanation['precision'])
print('\nExamples where anchor applies and model predicts %s:' % pred)
print('\n'.join([x[0] for x in explanation['raw']['examples'][-1]['covered_true']]))
print('\nExamples where anchor applies and model predicts %s:' % alternative)
print('\n'.join([x[0] for x in explanation['raw']['examples'][-1]['covered_false']]))
```

```
Anchor: good AND book
Precision: 0.95

Examples where anchor applies and model predicts positive:
Another includes both good book .
Any explains that good book .
SOME refers another good book .
This makes an good book .
SOME encapsulates every good book .
That consists this good book .
THE carries this good book .
Both is another good book .
Every sits another good book .
BOTH leads the good book .

Examples where anchor applies and model predicts negative:
SOME falls another good book .
THE feels another good book .
This starts some good book .
THis starts another good book .
All requires a good book .
Some goes a good book .
This happens some good book .
Some starts a good book .
Both feels some good book .
Both feels another good book .
```

CHAPTER 12

Anchor explanations for ImageNet

[1]:
```
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import keras
from keras.applications.inception_v3 import InceptionV3, preprocess_input, decode_
↪predictions
from alibi.datasets import imagenet
from alibi.explainers import AnchorImage
```

```
Using TensorFlow backend.
```

12.1 Load InceptionV3 model pre-trained on ImageNet

[2]:
```
model = InceptionV3(weights='imagenet')
```

12.2 Download and preprocess some images from ImageNet

The *imagenet* function takes as arguments one of the following ImageNet categories: 'Persian cat', 'volcano', 'strawberry', 'jellyfish' or 'centipede' as well as the number of images to return and the target size of the image.

[3]:
```
category = 'Persian cat'
image_shape = (299, 299, 3)
data, labels = imagenet(category, nb_images=10, target_size=image_shape[:2], seed=2)
print('Images shape: {}'.format(data.shape))
```

```
Images shape: (10, 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])
```

```
[('n02123394', 'Persian_cat', 0.9637921), ('n07615774', 'ice_lolly', 0.0015529424),␣
↪('n03207941', 'dishwasher', 0.0012963532)]
```

12.3 Define prediction function

```
[5]: predict_fn = lambda x: model.predict(x)
```

12.4 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.

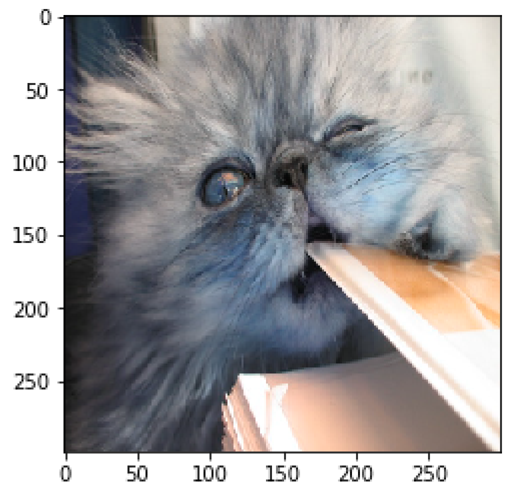
```
[6]: segmentation_fn = 'slic'
     kwargs = {'n_segments': 15, 'compactness': 20, 'sigma': .5}
     explainer = AnchorImage(predict_fn, image_shape, segmentation_fn=segmentation_fn,
                             segmentation_kwargs=kwargs, images_background=None)
```

12.5 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])
```

```
[7]: <matplotlib.image.AxesImage at 0x7f77214b0dd8>
```



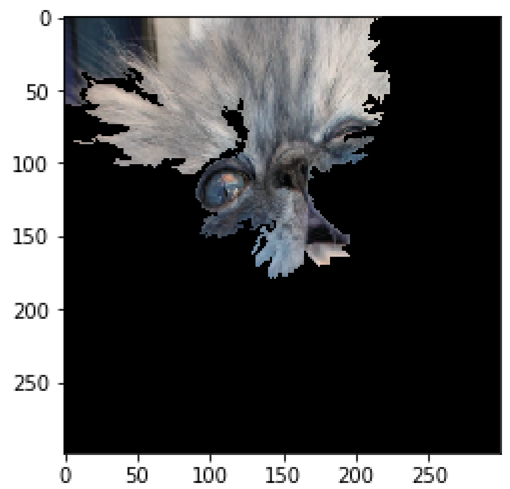
The *threshold*, *p_sample* and *tau* parameters are also key to generate a sensible explanation and ensure fast enough convergence. The *threshold* defines the minimum fraction of samples for a candidate anchor that need to lead to the same prediction as the original instance. While a higher threshold 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)
```

Superpixels in the anchor:

```
[9]: plt.imshow(explanation['anchor'])
```

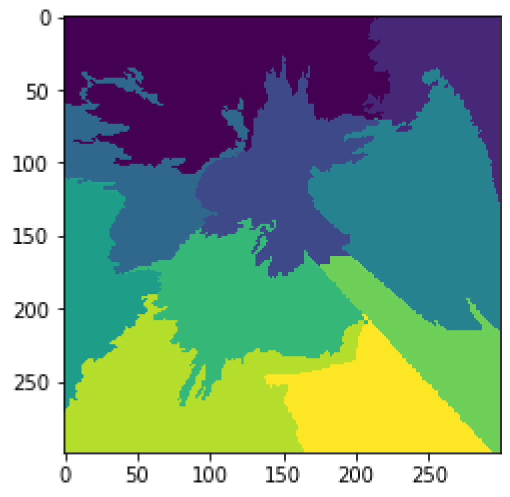
```
[9]: <matplotlib.image.AxesImage at 0x7f7720ae6f98>
```



A visualization of all the superpixels:

```
[10]: plt.imshow(explanation['segments'])
```

```
[10]: <matplotlib.image.AxesImage at 0x7f7720b98ba8>
```



CHAPTER 13

Anchor explanations for fashion MNIST

```
[1]: import matplotlib
     %matplotlib inline
     import matplotlib.pyplot as plt
     import numpy as np
     import keras
     from keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input
     from keras.models import Model
     from keras.utils import to_categorical
     from alibi.explainers import AnchorImage
```

```
Using TensorFlow backend.
```

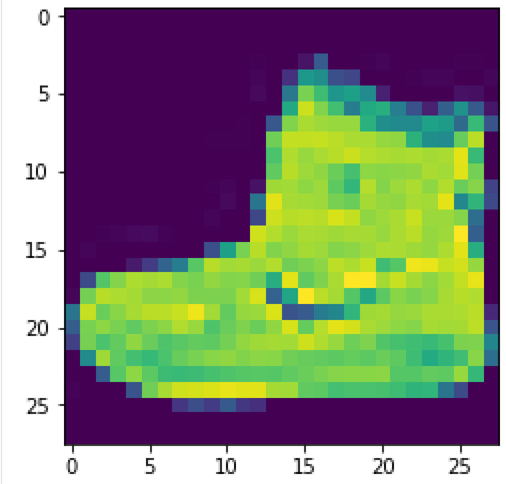
13.1 Load and prepare fashion MNIST data

```
[2]: (x_train, y_train), (x_test, y_test) = 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])
```

```
[3]: <matplotlib.image.AxesImage at 0x7fc93802a320>
```



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)
```

13.2 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()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 28, 28, 1)         0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 64)        320       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 64)        0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 32)        8224      
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 32)          0         
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 32)          0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 1568)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 256)               401664    
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570      
=================================================================
Total params: 412,778
Trainable params: 412,778
Non-trainable params: 0
_________________________________________________________________
```

13.3 Train model

[7]:
```
cnn.fit(x_train, y_train, batch_size=64, epochs=3)
```

```
Epoch 1/3
60000/60000 [==============================] - 40s 672us/step - loss: 0.5861 - acc: 0.
↪7857
Epoch 2/3
60000/60000 [==============================] - 40s 659us/step - loss: 0.4079 - acc: 0.
↪8522
Epoch 3/3
60000/60000 [==============================] - 42s 703us/step - loss: 0.3647 - acc: 0.
↪8667
```

[7]:
```
<keras.callbacks.History at 0x7fc9383eb7b8>
```

[8]:
```
# Evaluate the model on test set
score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])
```

```
Test accuracy:  0.8835
```

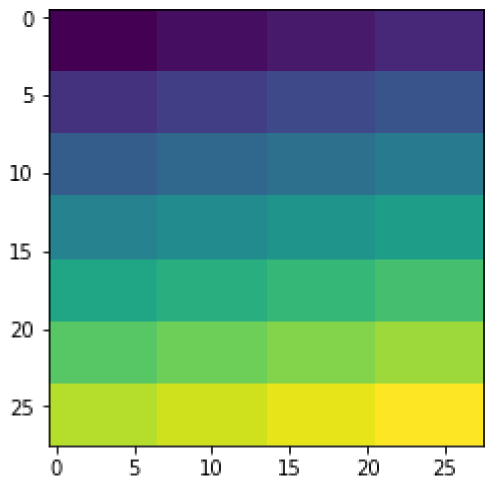
13.4 Define superpixels

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)
```

```
[10]: <matplotlib.image.AxesImage at 0x7fc9247d5ac8>
```



13.5 Define prediction function

```
[11]: predict_fn = lambda x: cnn.predict(x)
```

13.6 Initialize anchor image explainer

```
[12]: image_shape = x_train[idx].shape
      explainer = AnchorImage(predict_fn, image_shape, segmentation_fn=superpixel)
```

```
Specified both a segmentation function to create superpixels and keyword arguments␣
↪for built segmentation functions. By default the specified segmentation function␣
↪will be used.
```

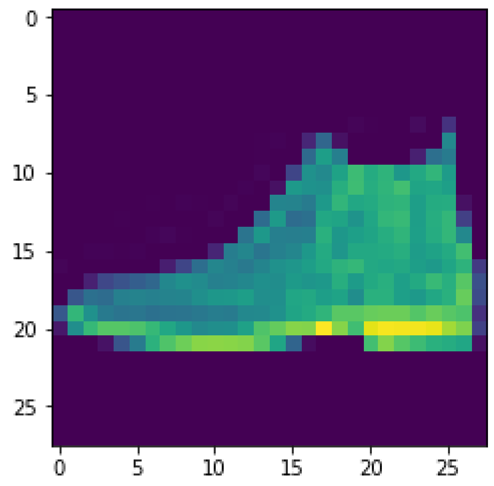
13.7 Explain a prediction

The explanation returns a mask with the superpixels that constitute the anchor.

Image to be explained:

```
[13]: i = 0
      image = x_test[i]
      plt.imshow(image[:,:,0])
```

```
[13]: <matplotlib.image.AxesImage at 0x7fc92c087048>
```



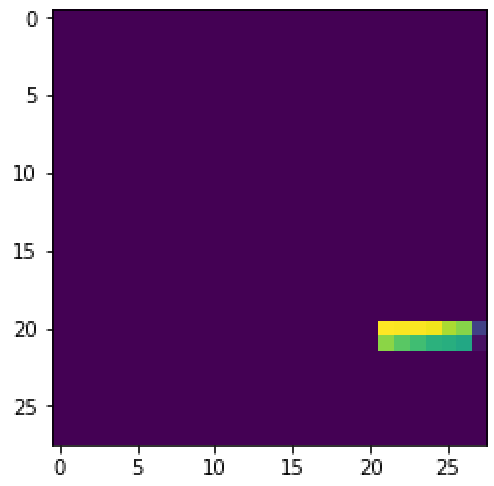
Generate explanation:

```
[14]: explanation = explainer.explain(image, threshold=.95, p_sample=.8)
```

Show anchor:

```
[15]: plt.imshow(explanation['anchor'][:,:,0])
```

```
[15]: <matplotlib.image.AxesImage at 0x7fc92c0c0ef0>
```



From the example, it looks like the heel alone is sufficient to predict a shoe.

CHAPTER 14

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.

```
[1]: import keras
     from keras import backend as K
     from keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input,␣
      ↪UpSampling2D
     from keras.models import Model, load_model
     from keras.utils import to_categorical
     import matplotlib
     %matplotlib inline
     import matplotlib.pyplot as plt
     import numpy as np
     import os
     import tensorflow as tf
     from alibi.explainers import CEM
```

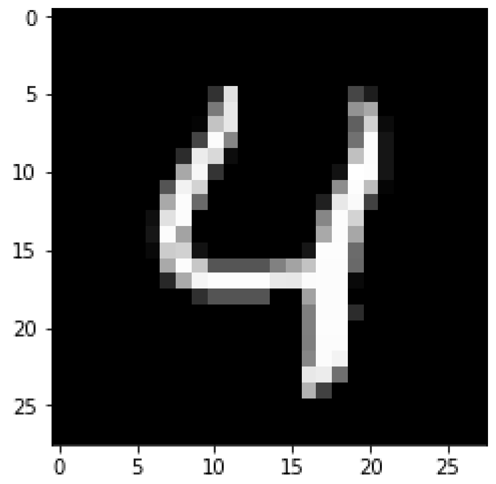
```
Using TensorFlow backend.
```

14.1 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,)
```

```
[2]: <matplotlib.image.AxesImage at 0x7f36b0f394e0>
```



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
```

14.2 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)
```

(continues on next page)

(continued from previous page)

```
    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=0)
cnn.save('mnist_cnn.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 64)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 32)        8224
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 32)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 32)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1568)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               401664
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 412,778
Trainable params: 412,778
Non-trainable params: 0
_________________________________________________________________
```

Evaluate the model on test set

[7]:
```
score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])
```

```
Test accuracy:  0.9892
```

14.3 Define and train auto-encoder

[8]:
```
def ae_model():
    x_in = Input(shape=(28, 28, 1))
```

(continues on next page)

(continued from previous page)

```
    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=10, validation_data=(x_test, x_test),␣
↪verbose=0)
ae.save('mnist_ae.h5')
```

```
_________________________________________________________________
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 (MaxPooling2 (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 14, 14, 1)         145
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 14, 14, 16)        160
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 28, 28, 16)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 28, 28, 16)        2320
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 28, 28, 1)         145
=================================================================
Total params: 5,250
Trainable params: 5,250
Non-trainable params: 0
_________________________________________________________________
```

Compare original with decoded images

[10]:
```
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)
```

(continues on next page)

(continued from previous page)

```
    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()
```

14.4 Generate contrastive explanation with pertinent negative

Explained instance:

```
[11]: idx = 4
X = x_test[idx].reshape((1,) + x_test[idx].shape)
```

CEM parameters:

```
[12]: 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:

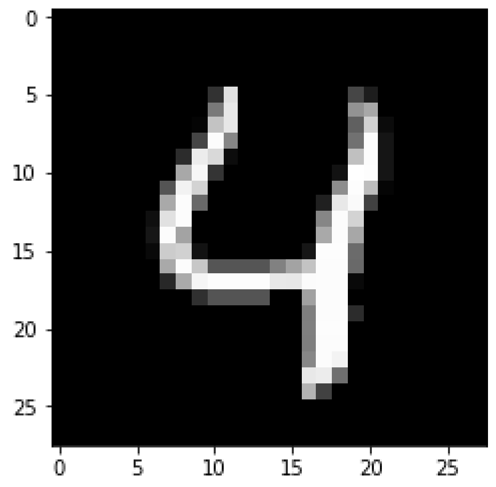
```
[13]: # initialize TensorFlow session before model definition
sess = tf.Session()
K.set_session(sess)
sess.run(tf.global_variables_initializer())

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

# initialize CEM explainer and explain instance
cem = CEM(sess, 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, verbose=False)

sess.close()
K.clear_session()
```

Original instance and prediction:

```
[14]: print('Original instance prediction: {}'.format(explanation['X_pred']))
plt.imshow(explanation['X'].reshape(28, 28))
```

```
Original instance prediction: 4
```

```
[14]: <matplotlib.image.AxesImage at 0x7f36453d0828>
```

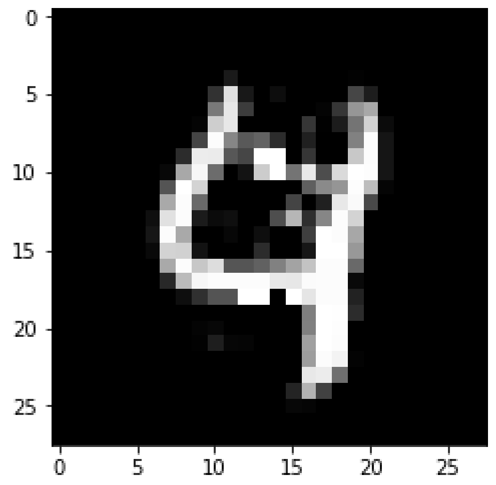


Pertinent negative:

```
[15]: print('Pertinent negative prediction: {}'.format(explanation[mode + '_pred']))
plt.imshow(explanation[mode].reshape(28, 28))
```

```
Pertinent negative prediction: 9
```

```
[15]: <matplotlib.image.AxesImage at 0x7f36453b5400>
```



14.5 Generate pertinent positive

[16]:
```
mode = 'PP'
```

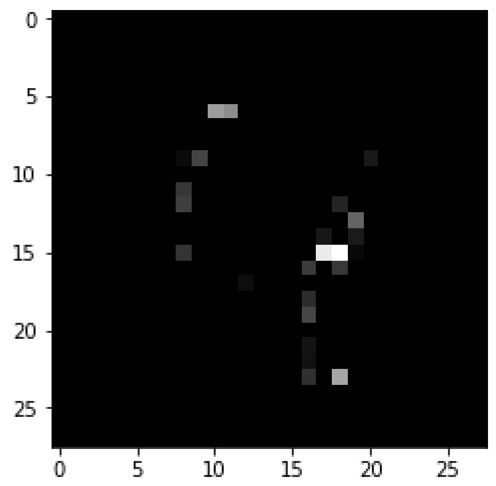
[17]:
```
# initialize TensorFlow session before model definition
sess = tf.Session()
K.set_session(sess)
sess.run(tf.global_variables_initializer())

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

# initialize CEM explainer and explain instance
cem = CEM(sess, 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, verbose=False)

sess.close()
K.clear_session()
```

Pertinent positive:

[18]:
```
print('Pertinent positive prediction: {}'.format(explanation[mode + '_pred']))
plt.imshow(explanation[mode].reshape(28, 28))
```

```
Pertinent positive prediction: 4
```

[18]:
```
<matplotlib.image.AxesImage at 0x7f363fa95588>
```



Clean up:

```
[19]: os.remove('mnist_cnn.h5')
      os.remove('mnist_ae.h5')
```

CHAPTER 15

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.

[1]:
```
import keras
from keras import backend as K
from keras.layers import Dense, Input
from keras.models import Model, load_model
from 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
import tensorflow as tf
from alibi.explainers import CEM
```

```
Using TensorFlow backend.
```

15.1 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)
```

15.2 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=128, epochs=500, verbose=0)
     lr.save('iris_lr.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 4)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 15
=================================================================
Total params: 15
Trainable params: 15
Non-trainable params: 0
_________________________________________________________________
```

15.3 Generate contrastive explanation with pertinent negative

Explained instance:

```
[7]: idx = 0
     X = x_test[idx].reshape((1,) + x_test[idx].shape)
     print('Prediction on instance to be explained: {}'.format(class_names[np.argmax(lr.
     ↪predict(X))]))
     print('Prediction probabilities for each class on the instance: {}'.format(lr.
     ↪predict(X)))
```

```
Prediction on instance to be explained: versicolor
Prediction probabilities for each class on the instance: [[0.00758386 0.53759295 0.
↪4548232 ]]
```

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]: # init session before model definition
sess = tf.Session()
K.set_session(sess)
sess.run(tf.global_variables_initializer())

# define model
lr = load_model('iris_lr.h5')

# initialize CEM explainer and explain instance
cem = CEM(sess, 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)
sess.close()
K.clear_session()
```

```
[10]: print('Original instance: {}'.format(explanation['X']))
print('Predicted class: {}'.format(class_names[explanation['X_pred']]))
```

```
Original instance: [[ 0.55333328 -1.28296331  0.70592084  0.92230284]]
Predicted class: versicolor
```

```
[11]: print('Pertinent negative: {}'.format(explanation[mode]))
print('Predicted class: {}'.format(class_names[explanation[mode + '_pred']]))
```

```
Pertinent negative: [[ 0.5533333 -1.2829633  1.0322616  1.0454441]]
Predicted class: virginica
```

Store explanation to plot later on:

[12]:
```
expl = {}
expl['PN'] = explanation[mode]
expl['PN_pred'] = explanation[mode + '_pred']
```

15.4 Generate pertinent positive

[13]:
```
mode = 'PP'
```

Generate pertinent positive:

[14]:
```
# init session before model definition
sess = tf.Session()
K.set_session(sess)
sess.run(tf.global_variables_initializer())

# define model
lr = load_model('iris_lr.h5')

# initialize CEM explainer and explain instance
cem = CEM(sess, 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)
sess.close()
K.clear_session()
```

[15]:
```
print('Pertinent positive: {}'.format(explanation[mode]))
print('Predicted class: {}'.format(class_names[explanation[mode + '_pred']]))
```

```
Pertinent positive: [[-7.44469730e-09 -3.47054341e-08  2.67991149e-08 -4.
↪21282409e-09]]
Predicted class: versicolor
```

[16]:
```
expl['PP'] = explanation[mode]
expl['PP_pred'] = explanation[mode + '_pred']
```

15.5 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']]
```

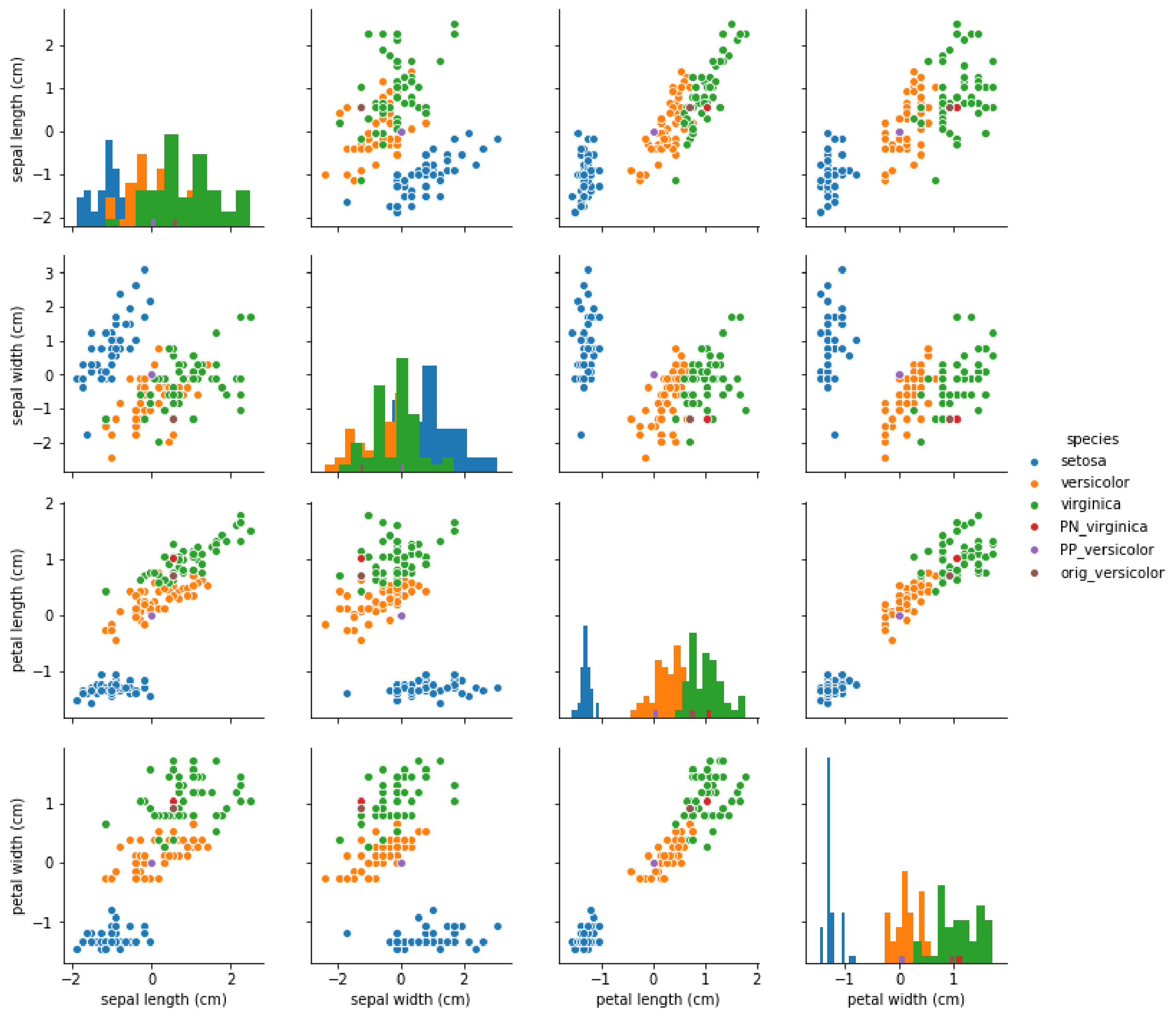
(continues on next page)

(continued from previous page)

```
orig_inst = pd.DataFrame(explanation['X'], columns=dataset.feature_names)
orig_inst['species'] = 'orig_' + class_names[explanation['X_pred']]
df = df.append([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');
```



15.6 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 x nb of prediction categories)* and for *eps[1]* it should be *(1 x nb 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]: # init session before model definition
      sess = tf.Session()
      K.set_session(sess)
      sess.run(tf.global_variables_initializer())

      # 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(sess, 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)
      sess.close()
      K.clear_session()
```

```
[24]: print('Original instance: {}'.format(explanation['X']))
      print('Predicted class: {}'.format(class_names[explanation['X_pred']]))
```

```
Original instance: [[ 0.55333328 -1.28296331  0.70592084  0.92230284]]
Predicted class: versicolor
```

```
[25]: print('Pertinent negative: {}'.format(explanation[mode]))
      print('Predicted class: {}'.format(class_names[explanation[mode + '_pred']]))
```

```
Pertinent negative: [[ 0.5533333 -1.2829633  1.0467366  1.0182681]]
Predicted class: virginica
```

Clean up:

```
[26]: os.remove('iris_lr.h5')
```

CHAPTER 16

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.

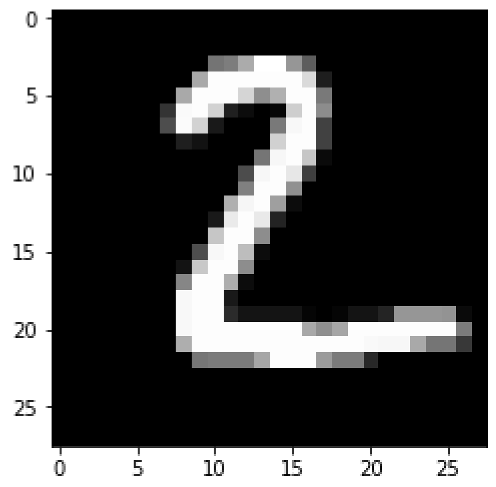
```
[1]: import keras
from keras import backend as K
from keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input
from keras.models import Model, load_model
from keras.utils import to_categorical
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
import tensorflow as tf
from time import time
from alibi.explainers import CounterFactual
```

```
Using TensorFlow backend.
```

16.1 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[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
```

16.2 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)
```

(continues on next page)

(continued from previous page)

```
    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=1)
cnn.save('mnist_cnn.h5')
```

```
WARNING:tensorflow:From /home/janis/.conda/envs/py36dev/lib/python3.6/site-packages/
↪tensorflow/python/framework/op_def_library.py:263: colocate_with (from tensorflow.
↪python.framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From /home/janis/.conda/envs/py36dev/lib/python3.6/site-packages/
↪keras/backend/tensorflow_backend.py:3445: calling dropout (from tensorflow.python.
↪ops.nn_ops) with keep_prob is deprecated and will be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 -␣
↪keep_prob`.
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 64)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 32)        8224
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 32)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 32)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1568)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               401664
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 412,778
Trainable params: 412,778
Non-trainable params: 0
```

(continues on next page)

(continued from previous page)

```
_________________________________________________________________
WARNING:tensorflow:From /home/janis/.conda/envs/py36dev/lib/python3.6/site-packages/
↪tensorflow/python/ops/math_ops.py:3066: to_int32 (from tensorflow.python.ops.
↪math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Epoch 1/3
60000/60000 [==============================] - 53s 875us/step - loss: 0.3319 - acc: 0.
↪8940
Epoch 2/3
60000/60000 [==============================] - 43s 712us/step - loss: 0.1187 - acc: 0.
↪9629
Epoch 3/3
60000/60000 [==============================] - 44s 733us/step - loss: 0.0930 - acc: 0.
↪9719
```

Evaluate the model on test set

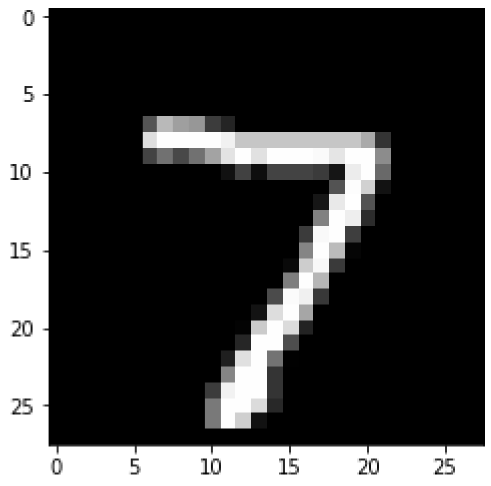
```
[7]: score = cnn.evaluate(x_test, y_test, verbose=0)
     print('Test accuracy: ', score[1])
```

```
Test accuracy:  0.987
```

16.3 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
```

(continues on next page)

(continued from previous page)

```
lam_init = 1e-1
max_lam_steps = 10
learning_rate_init = 0.1
feature_range = (x_train.min(),x_train.max())
```

Run counterfactual:

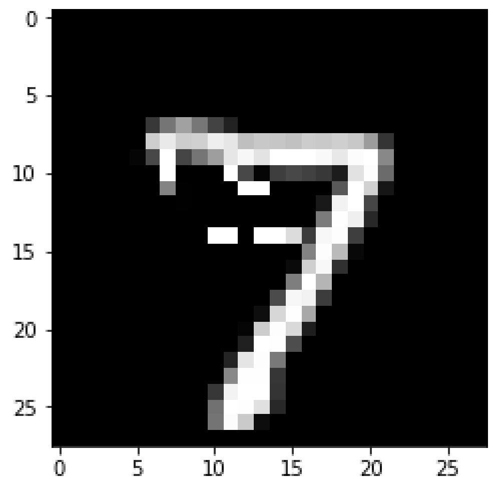
```
[10]: # set random seed
np.random.seed(1)
tf.set_random_seed(1)

sess = K.get_session()

# initialize explainer
cf = CounterFactual(sess, 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))
```

```
WARNING:tensorflow:From /home/janis/.conda/envs/py36dev/lib/python3.6/site-packages/
↪tensorflow/python/training/learning_rate_decay_v2.py:321: div (from tensorflow.
↪python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Deprecated in favor of operator or tf.math.divide.
Explanation took 6.444 sec
```

Results:

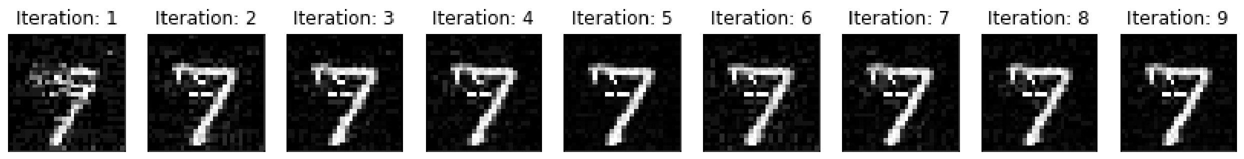
```
[11]: pred_class = explanation['cf']['class']
proba = explanation['cf']['proba'][0][pred_class]

print(f'Counterfactual prediction: {pred_class} with probability {proba}')
plt.imshow(explanation['cf']['X'].reshape(28, 28));
```

```
Counterfactual prediction: 9 with probability 0.9900996088981628
```



The counterfactual starting from a 7 moves towards the closest class as determined by the model and the data: 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']}
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:

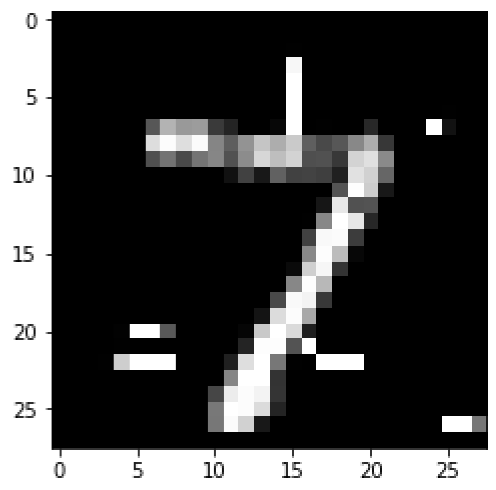
[13]:
```
target_class = 1

cf = CounterFactual(sess, 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 4.563 sec
```

Results:

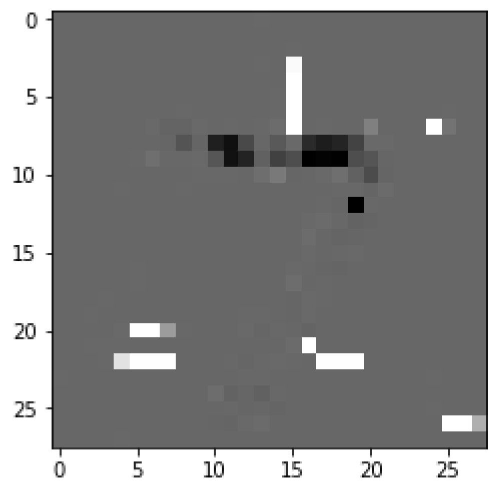
[14]:
```
pred_class = explanation['cf']['class']
proba = explanation['cf']['proba'][0][pred_class]

print(f'Counterfactual prediction: {pred_class} with probability {proba}')
plt.imshow(explanation['cf']['X'].reshape(28, 28));
```

```
Counterfactual prediction: 1 with probability 0.9983848333358765
```



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')
```

CHAPTER 17

Counterfactuals guided by prototypes on MNIST

This method 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 auto-encoder 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 would 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:

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

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 γL_{AE} is to penalize out-of-distribution counterfactuals while θL_{proto} guides the counterfactual to a prototype. When we only have acces to the model's predict function and cannot fully enjoy the benefits of automatic differentiation, the prototypes allow us to drop the predict function loss term and still generate reasonable counterfactuals. This drastically reduces the number of predict 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.

The different use cases highlighted above are illustrated throughout this notebook.

[1]:
```
import keras
from keras import backend as K
from keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input,␣
↪UpSampling2D
from keras.models import Model, load_model
from keras.utils import to_categorical
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
```

(continues on next page)

(continued from previous page)

```
import tensorflow as tf
from time import time
from alibi.explainers import CounterFactualProto
```

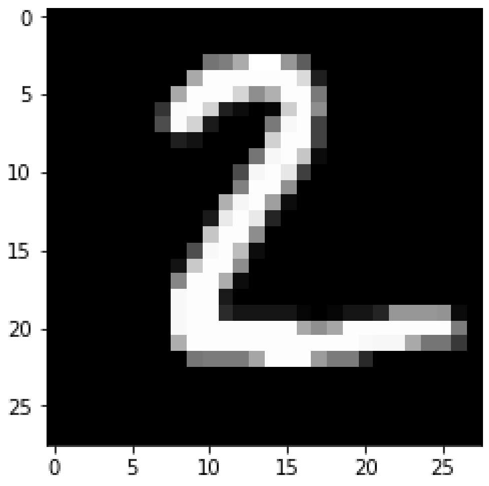
```
Using TensorFlow backend.
```

17.1 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[1])
```

```
x_train shape: (60000, 28, 28) y_train shape: (60000,)
```

[2]:
```
<matplotlib.image.AxesImage at 0x7fd0ba0f40b8>
```



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
```

17.2 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=1)
     cnn.save('mnist_cnn.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 28, 28, 1)         0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 64)        320       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 64)        0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 32)        8224      
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 32)          0         
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 32)          0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 1568)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 256)               401664    
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570      
=================================================================
Total params: 412,778
Trainable params: 412,778
Non-trainable params: 0
_________________________________________________________________
```

Evaluate the model on test set

```
[7]: score = cnn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])
```

```
Test accuracy:  0.9862
```

17.3 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.summary()
ae.fit(x_train, x_train, batch_size=128, epochs=4, validation_data=(x_test, x_test),␣
↪verbose=1)
ae.save('mnist_ae.h5')
enc.save('mnist_enc.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
model_1 (Model)              (None, 14, 14, 1)         2625
_________________________________________________________________
model_2 (Model)              (None, 28, 28, 1)         2625
=================================================================
Total params: 5,250
Trainable params: 5,250
Non-trainable params: 0
_________________________________________________________________
```

```
/home/avl/anaconda3/envs/alibi/lib/python3.6/site-packages/keras/engine/saving.
↪py:292: UserWarning: No training configuration found in save file: the model was␣
↪*not* compiled. Compile it manually.
```

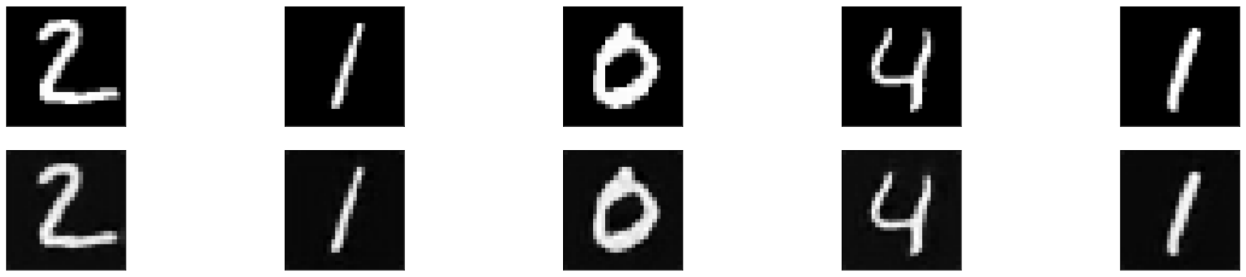
(continues on next page)

(continued from previous page)

```
warnings.warn('No training configuration found in save file: '
```

Compare original with decoded images

```
[10]: 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()
```

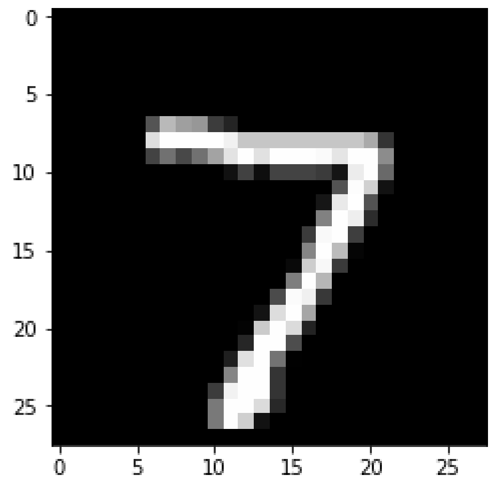


17.4 Generate counterfactual guided by the nearest class prototype

Original instance:

```
[11]: X = x_test[0].reshape((1,) + x_test[0].shape)
plt.imshow(X.reshape(28, 28))
```

```
[11]: <matplotlib.image.AxesImage at 0x7fd04d57c6d8>
```



Counterfactual parameters:

```
[12]: shape = (1,) + x_train.shape[1:]
gamma = 100.
theta = 100.
c_init = 1.
c_steps = 2
max_iterations = 500
feature_range = (x_train.min(),x_train.max())
```

Run counterfactual:

```
[13]: # set random seed
np.random.seed(1)
tf.set_random_seed(1)

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

sess = K.get_session()

# initialize explainer, fit and generate counterfactual
cf = CounterFactualProto(sess, 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))

sess.close()
K.clear_session()
```

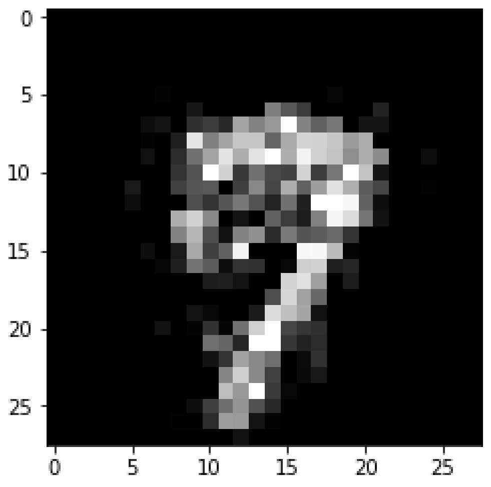
```
Time to find prototypes each class: 16.354 sec
Explanation took 4.252 sec
```

Results:

[14]:
```
print('Counterfactual prediction: {}'.format(explanation['cf']['class']))
print('Closest prototype class: {}'.format(cf.id_proto))
plt.imshow(explanation['cf']['X'].reshape(28, 28))
```

```
Counterfactual prediction: 9
Closest prototype class: 9
```

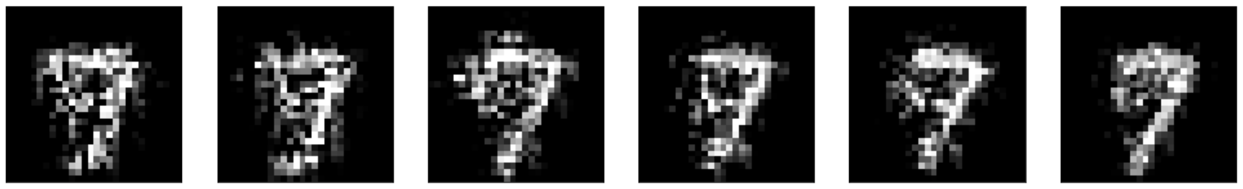
[14]:
```
<matplotlib.image.AxesImage at 0x7fd04bfb67b8>
```



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('iteration c {}'.format(iter_cf))
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.

17.5 Remove the auto-encoder loss term

In the previous example, we used both an auto-encoder 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 auto-encoder loss term to speed up the counterfactual search and still get decent results:

[16]:
```
theta = 500.
```

[17]:
```
# set random seed
np.random.seed(1)
tf.set_random_seed(1)

# define models
cnn = load_model('mnist_cnn.h5')
enc = load_model('mnist_enc.h5')

sess = K.get_session()

# initialize explainer, fit and generate counterfactual
cf = CounterFactualProto(sess, cnn, shape, 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)
print('Explanation took {:.3f} sec'.format(time() - start_time))

sess.close()
K.clear_session()
```

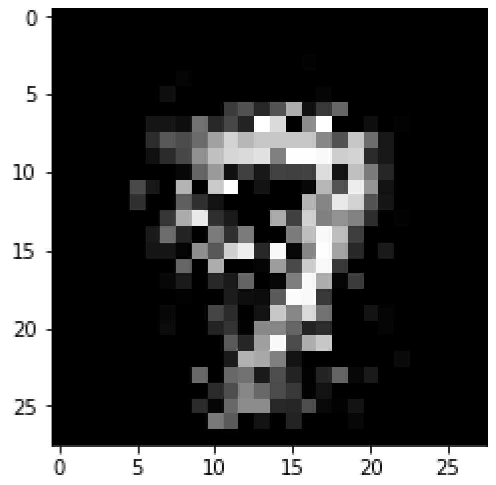
```
Explanation took 2.877 sec
```

Results:

[18]:
```
print('Counterfactual prediction: {}'.format(explanation['cf']['class']))
print('Closest prototype class: {}'.format(cf.id_proto))
plt.imshow(explanation['cf']['X'].reshape(28, 28))
```

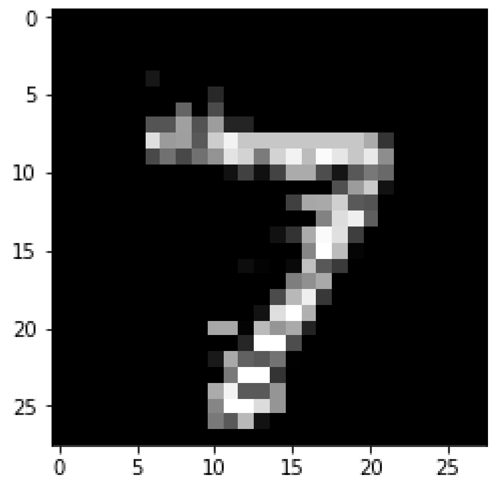
```
Counterfactual prediction: 9
Closest prototype class: 9
```

[18]:
```
<matplotlib.image.AxesImage at 0x7fd04a3d3fd0>
```

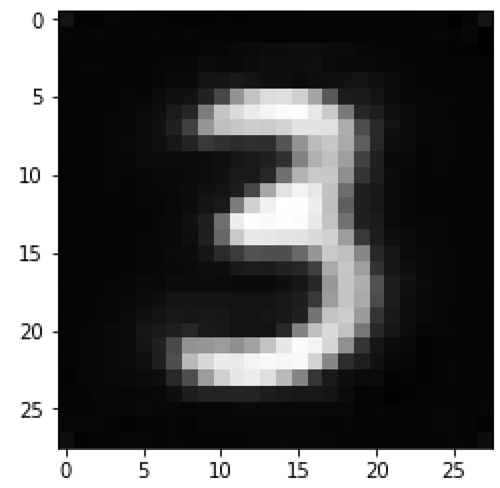


17.6 Add trust scores

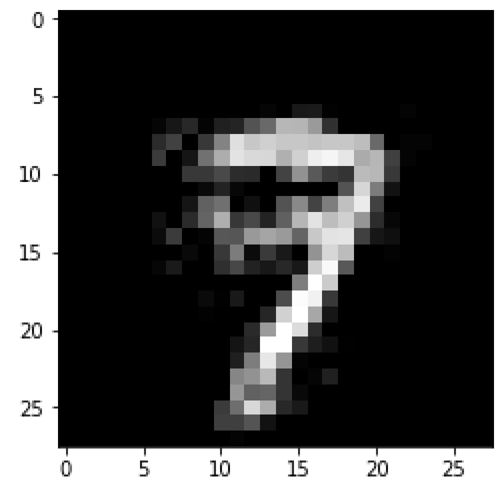
Explainability through counterfactuals can involve a trade off between sparsity and interpretability. The first counterfactual below is sparse and flips the original 7 into a 3. It is obtained by setting the auto-encoder and prototype loss terms to 0, moving the objective function towards an elastic net regularizer which generates sparse solutions. It is clear however that this is not a very interpretable 3 which lies very far from its class prototype or the training data distribution:



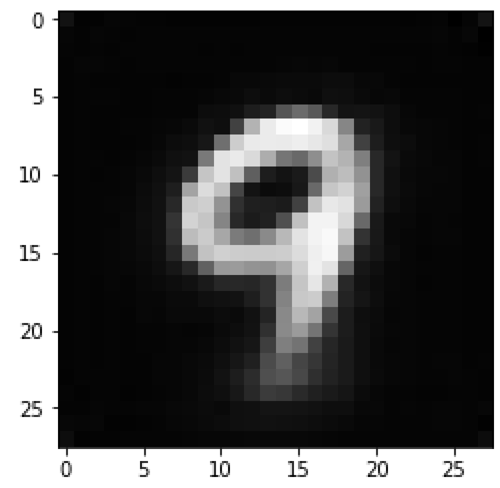
Class 3 prototype:



By adding in the nearest prototype and auto-encoder loss terms, the counterfactual becomes more interpretable but less sparse. The example below illustrates this by flipping the same 7 into a 9:



Class 9 prototype:



In order to help interpretability, we can 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*. The example below shows the impact of such a trust score constraint:

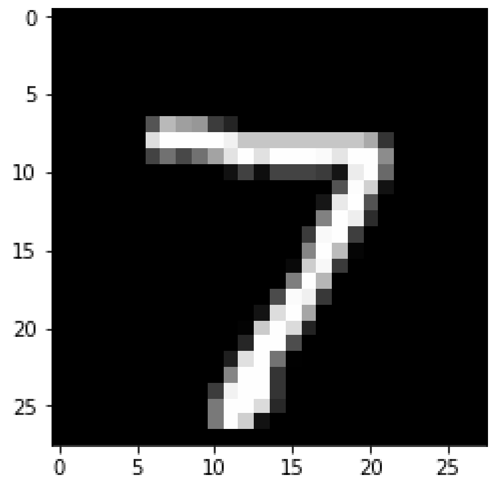
```
[19]: theta = 100.
      c_init = 1.
      c_steps = 5
```

```
[24]: # set random seed
      np.random.seed(1)
      tf.set_random_seed(1)

      # define models
      cnn = load_model('mnist_cnn.h5')
      enc = load_model('mnist_enc.h5')

      sess = K.get_session()

      # initialize explainer, fit and generate counterfactual
      cf = CounterFactualProto(sess, cnn, shape, 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)  # find class prototypes
      explanation1 = cf.explain(X, threshold=0.)
      explanation2 = cf.explain(X, threshold=.8)

      sess.close()
      K.clear_session()
```

Original instance:

```
[25]: plt.imshow(X.reshape(28, 28))
```

```
[25]: <matplotlib.image.AxesImage at 0x7fd0431fca90>
```

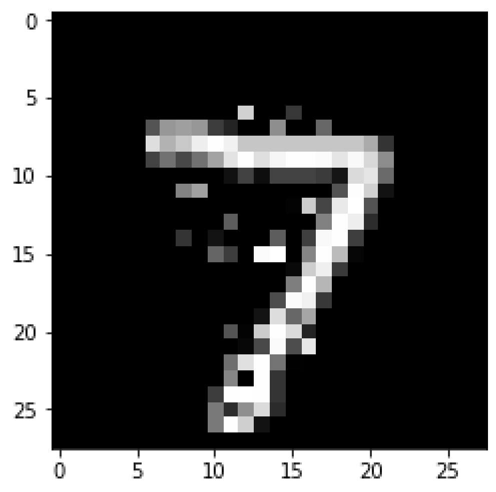


The counterfactual below without the trust score threshold predicts a 3 which actually starts to look like a 9 since that is the nearest prototype class. This counterfactual is slightly sparser than the second counterfactual below which has the trust score threshold at 0.8. The sparser counterfactual does not meet this trust score constraint and is therefore rejected in the second example.

```
[26]: print('Counterfactual prediction: {}'.format(explanation1['cf']['class']))
print('Closest prototype class: {}'.format(cf.id_proto))
plt.imshow(explanation1['cf']['X'].reshape(28, 28))
```

```
Counterfactual prediction: 3
Closest prototype class: 9
```

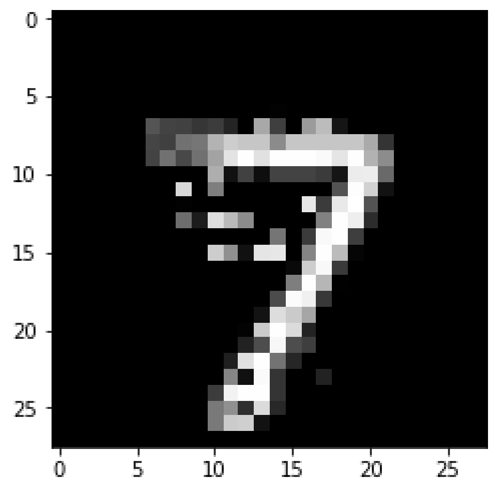
```
[26]: <matplotlib.image.AxesImage at 0x7fd04315a710>
```



```
[27]: print('Counterfactual prediction: {}'.format(explanation2['cf']['class']))
print('Closest prototype class: {}'.format(cf.id_proto))
plt.imshow(explanation2['cf']['X'].reshape(28, 28))
```

```
Counterfactual prediction: 9
Closest prototype class: 9
```

```
[27]: <matplotlib.image.AxesImage at 0x7fd043130dd8>
```

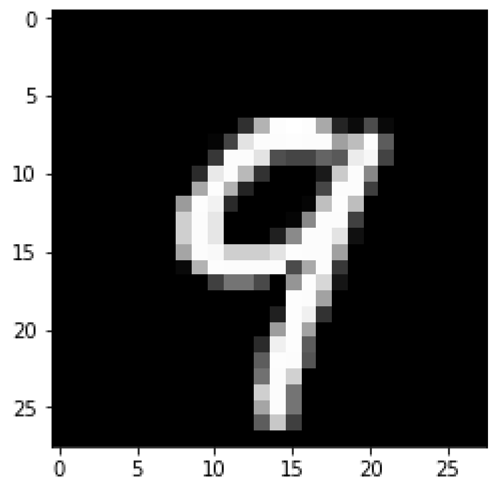


17.7 Specify prototype classes

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. It does not guarantee that the avoided classes are excluded from the counterfactuals, but does not include those classes when looking for the nearest prototype class. So as a result, it will not drive the counterfactual to the prototype of the classes that should be avoided.

[28]:
```
X = x_test[12].reshape((1,) + x_test[1].shape)
plt.imshow(X.reshape(28, 28))
```

[28]:
```
<matplotlib.image.AxesImage at 0x7fd04308e240>
```



[29]:
```
theta = 100.
gamma = 100.
c_init = 1.
c_steps = 3
max_iterations = 1000
```

[30]:
```
# set random seed
np.random.seed(1)
tf.set_random_seed(1)

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

sess = K.get_session()

# initialize explainer, fit and generate counterfactual
cf = CounterFactualProto(sess, 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)  # find class prototypes
explanation1 = cf.explain(X)
proto_class1 = cf.id_proto
explanation2 = cf.explain(X, target_class=[7])
proto_class2 = cf.id_proto
```

(continues on next page)

(continued from previous page)

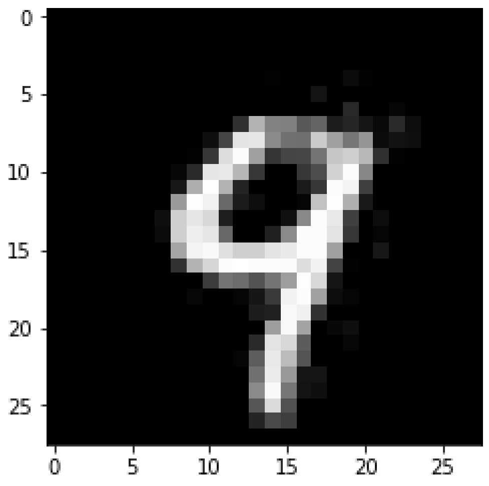
```
sess.close()
K.clear_session()
```

The closest class to a 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, the 9 is not pushed towards a 4.

```
[31]: print('Counterfactual prediction: {}'.format(explanation1['cf']['class']))
print('Closest prototype class: {}'.format(proto_class1))
plt.imshow(explanation1['cf']['X'].reshape(28, 28))
```

```
Counterfactual prediction: 4
Closest prototype class: 4
```

```
[31]: <matplotlib.image.AxesImage at 0x7fd048f87ef0>
```

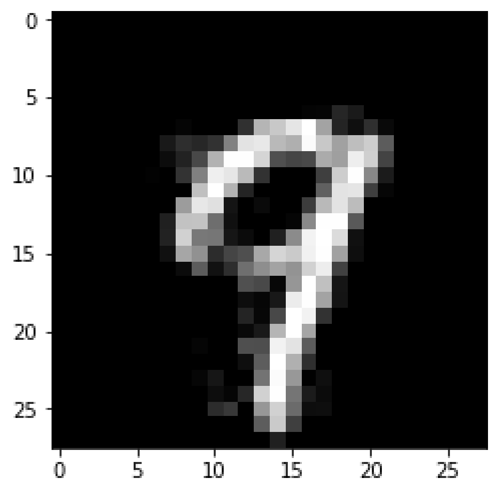


While the counterfactual still looks like a 9 at first sight, it is clear by looking at the difference plot between the counterfactual and the original instance that the pixels generating the circle of the 9 are being removed, moving towards a 7.

```
[32]: print('Counterfactual prediction: {}'.format(explanation2['cf']['class']))
print('Closest prototype class: {}'.format(proto_class2))
plt.imshow(explanation2['cf']['X'].reshape(28, 28))
```

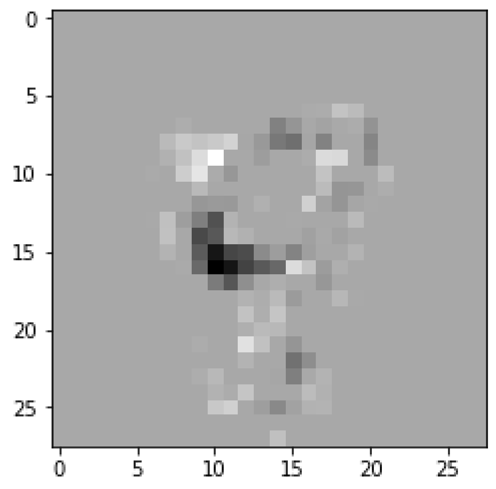
```
Counterfactual prediction: 7
Closest prototype class: 7
```

```
[32]: <matplotlib.image.AxesImage at 0x7fd0485354e0>
```



```
[33]: plt.imshow((explanation2['cf']['X'] - X).reshape(28, 28))
```

```
[33]: <matplotlib.image.AxesImage at 0x7fd0493b3c88>
```



17.8 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 predict function, the gradient updates are numerical and typically require a large number of prediction calls because of the predict loss term. These prediction calls can slow the search down significantly and become a bottleneck. We can represent the gradient of the loss term as follows:

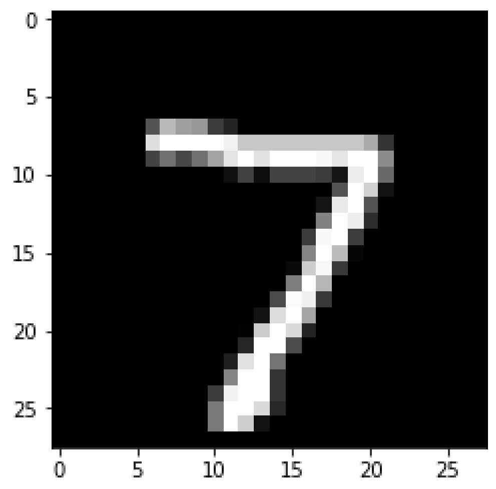
$$\frac{\partial L_{pred}}{\partial x} = \frac{\partial L_{pred}}{\partial p}\frac{\partial p}{\partial x}$$

where L_{pred} is the prediction loss term, p the predict 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 28x28x2 = 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.

We'll first illustrate this method by using a Keras model and then try a black box model.

[34]:
```
X = x_test[0].reshape((1,) + x_test[0].shape)
plt.imshow(X.reshape(28, 28))
```

[34]:
```
<matplotlib.image.AxesImage at 0x7fd04d26f128>
```



[35]:
```
c_init = 0.  # set weight predict loss term to 0
c_steps = 1  # do not update c further
max_iterations = 2000
theta = 100.
gamma = 100.
```

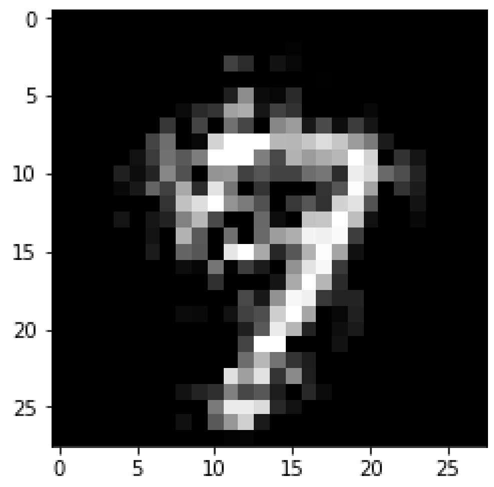
[36]:
```
# set random seed
np.random.seed(1)
tf.set_random_seed(1)

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

sess = K.get_session()

# initialize explainer, fit and generate counterfactual
cf = CounterFactualProto(sess, 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)  # find class prototypes
explanation = cf.explain(X)

sess.close()
K.clear_session()
```

Still get a reasonable 9:

```
[37]: print('Counterfactual prediction: {}'.format(explanation['cf']['class']))
print('Closest prototype class: {}'.format(cf.id_proto))
plt.imshow(explanation['cf']['X'].reshape(28, 28))
```

```
Counterfactual prediction: 9
Closest prototype class: 9
```

```
[37]: <matplotlib.image.AxesImage at 0x7fd04d689438>
```



Let us now use the black box model:

```
[38]: # set random seed
np.random.seed(1)
tf.set_random_seed(1)

# define models
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')

sess = K.get_session()

# initialize explainer, fit and generate counterfactual
cf = CounterFactualProto(sess, 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)  # find class prototypes
start_time = time()
explanation = cf.explain(X)
print('Explanation took {:.3f} sec'.format(time() - start_time))

sess.close()
K.clear_session()
```

```
Explanation took 8.979 sec
```

Which again gives a 9 as counterfactual:

```
[39]: print('Counterfactual prediction: {}'.format(explanation['cf']['class']))
print('Closest prototype class: {}'.format(cf.id_proto))
```

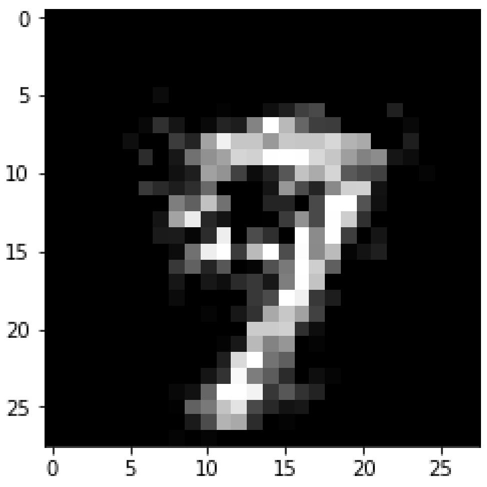
(continues on next page)

(continued from previous page)

```
plt.imshow(explanation['cf']['X'].reshape(28, 28))
```

```
Counterfactual prediction: 9
Closest prototype class: 9
```

[39]:
```
<matplotlib.image.AxesImage at 0x7fd04213bd30>
```



We can include the prediction loss term again and compare the explanation time for the same amount of iteration steps:

[40]:
```
c_init = 1.  # set weight predict loss term to 0
c_steps = 4  # do not update c further
max_iterations = 500  # 4x500 = 2000
```

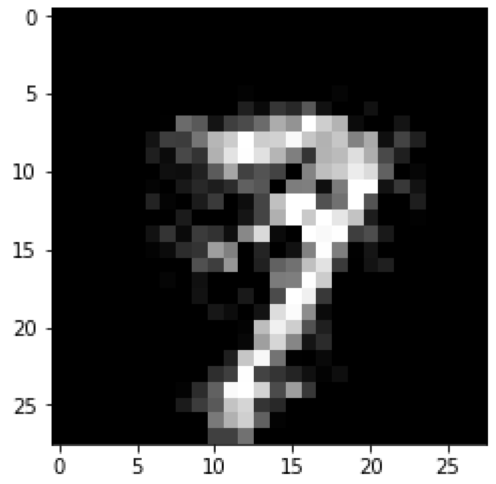
[41]:
```
# set random seed
np.random.seed(1)
tf.set_random_seed(1)

# define models
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')

sess = K.get_session()

# initialize explainer, fit and generate counterfactual
cf = CounterFactualProto(sess, 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)  # find class prototypes
start_time = time()
explanation = cf.explain(X)
print('Explanation took {:.3f} sec'.format(time() - start_time))

sess.close()
K.clear_session()
```

```
Explanation took 893.747 sec
```

```
[42]: print('Counterfactual prediction: {}'.format(explanation['cf']['class']))
      print('Closest prototype class: {}'.format(cf.id_proto))
      plt.imshow(explanation['cf']['X'].reshape(28, 28))
```

```
Counterfactual prediction: 9
Closest prototype class: 9
```

```
[42]: <matplotlib.image.AxesImage at 0x7fd04b731400>
```



By removing the predict loss term and having the nearest class prototype guide the counterfactual, we can speed up the search by about 100x for a simple CNN. Please note that these examples are not optimized for speed but just highlight the difference in performance.

Clean up:

```
[43]: os.remove('mnist_cnn.h5')
      os.remove('mnist_ae.h5')
      os.remove('mnist_enc.h5')
```

CHAPTER 18

Counterfactuals guided by prototypes on Boston 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 the Boston area 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.

[1]:
```
import keras
from keras import backend as K
from keras.layers import Dense, Input
from keras.models import Model, load_model
from keras.utils import to_categorical
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
from sklearn.datasets import load_boston
import tensorflow as tf
from alibi.explainers import CounterFactualProto
```

```
Using TensorFlow backend.
```

18.1 Load and prepare Boston housing dataset

[2]:
```
boston = load_boston()
data = boston['data']
target = boston['target']
feature_names = boston['feature_names']
```

Transform into classification task: target becomes whether house price is above the overall median or not

```
[3]: y = np.zeros((target.shape[0],))
     y[np.where(target > np.median(target))[0]] = 1
```

Remove categorical feature

```
[4]: data = np.delete(data, 3, 1)
     feature_names = np.delete(feature_names, 3)
```

Explanation of remaining features:

- CRIM: per capita crime rate by town
- ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS: proportion of non-retail business acres per town
- RM: average number of rooms per dwelling
- AGE: proportion of owner-occupied units built prior to 1940
- DIS: weighted distances to five Boston employment centres
- RAD: index of accessibility to radial highways
- TAX: full-value property-tax rate per USD10,000
- PTRATIO: pupil-teacher ratio by town
- B: 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT: % lower status of the population

Standardize data

```
[5]: mu = data.mean(axis=0)
     sigma = data.std(axis=0)
     data = (data - mu) / sigma
```

Define train and test set

```
[6]: idx = 475
     x_train,y_train = data[:idx,:], y[:idx]
     x_test, y_test = data[idx:,:], y[idx:]
     y_train = to_categorical(y_train)
     y_test = to_categorical(y_test)
```

18.2 Train model

```
[7]: np.random.seed(0)
     tf.set_random_seed(0)
```

```
[8]: def nn_model():
         x_in = Input(shape=(12,))
         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
```

[9]:
```
nn = nn_model()
nn.summary()
nn.fit(x_train, y_train, batch_size=128, epochs=500, verbose=0)
nn.save('nn_boston.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 12)                0
_________________________________________________________________
dense_1 (Dense)              (None, 40)                520
_________________________________________________________________
dense_2 (Dense)              (None, 40)                1640
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 82
=================================================================
Total params: 2,242
Trainable params: 2,242
Non-trainable params: 0
_________________________________________________________________
```

[10]:
```
score = nn.evaluate(x_test, y_test, verbose=0)
print('Test accuracy: ', score[1])
```

```
Test accuracy:  0.8064516186714172
```

18.3 Generate counterfactual guided by the nearest class prototype

Original instance:

[11]:
```
X = x_test[1].reshape((1,) + x_test[1].shape)
shape = X.shape
```

Run counterfactual:

[12]:
```
# set random seed
np.random.seed(1)
tf.set_random_seed(1)

# define model
nn = load_model('nn_boston.h5')

# get Keras session
sess = K.get_session()

# initialize explainer, fit and generate counterfactual
cf = CounterFactualProto(sess, 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)
explanation = cf.explain(X)

sess.close()
K.clear_session()
```

```
No encoder specified. Using k-d trees to represent class prototypes.
```

The prediction flipped from 0 (value below the median) to 1 (above the median):

```
[13]: print('Original prediction: {}'.format(explanation['orig_class']))
print('Counterfactual prediction: {}'.format(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:

```
[14]: 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('{}: {}'.format(f, delta[0][i]))
```

```
AGE: -21.56911332760376
LSTAT: -4.1864947634051575
```

So in order to increase the house price, the proportion of owner-occupied units built prior to 1940 should decrease by almost 22%. This is not surprising since the proportion for the observation is very high at 93.6%. Furthermore, the % of the population with "lower status" should decrease by 4%.

```
[15]: print('% owner-occupied units built prior to 1940: {}'.format(orig[0][5]))
print('% lower status of the population: {}'.format(orig[0][11]))
```

```
% owner-occupied units built prior to 1940: 93.6
% lower status of the population: 18.68
```

Clean up:

```
[16]: os.remove('boston_nn.h5')
```

CHAPTER 19

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
```

19.1 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:]
```

19.2 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('Predicted class: {}'.format(y_pred))
```

```
Predicted class: [2 2 2 2 2 2 2 2 2]
```

19.3 Basic Trust Score Usage

19.3.1 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 kth nearest neighbor of each tree (or the average distance to the kth 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
     ↪'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
```

19.3.2 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
```

(continues on next page)

(continued from previous page)

```
print('Trust scores: {}'.format(score))
print('\nClosest not predicted class: {}'.format(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]
```

19.4 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:
```

(continues on next page)

(continued from previous page)

```
        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_lr(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]
        # 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)):
```

(continues on next page)

(continued from previous page)

```
            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)
```

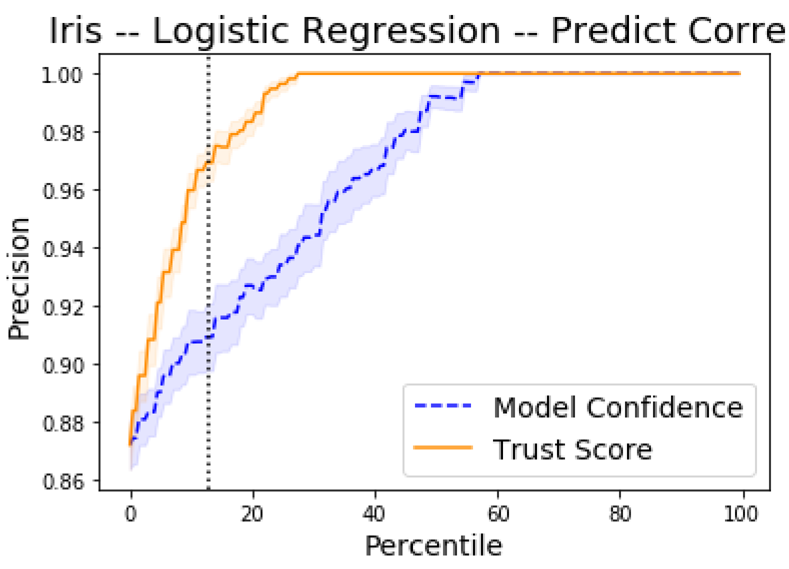
19.4.1 Detect correctly classified examples

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)]
nfolds = 10
plt_names = ['Model Confidence', 'Trust Score']
plt_title = 'Iris -- Logistic Regression -- Predict Correct'
```

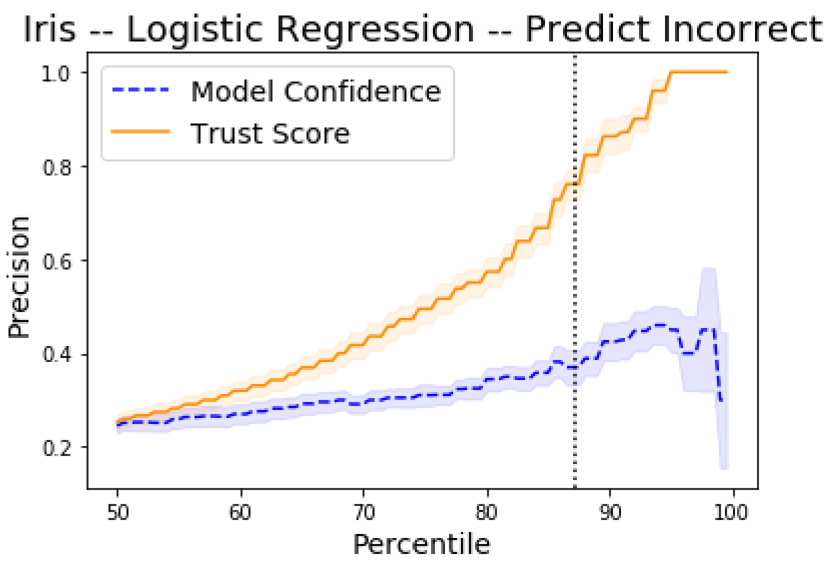
[13]:
```
run_precision_plt(X, y, nfolds, percentiles, run_lr, plt_title=plt_title,
                  plt_names=plt_names, predict_correct=True)
```



19.4.2 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, nfolds, percentiles, run_lr, plt_title=plt_title,
                        plt_names=plt_names, predict_correct=False)
```



CHAPTER 20

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.

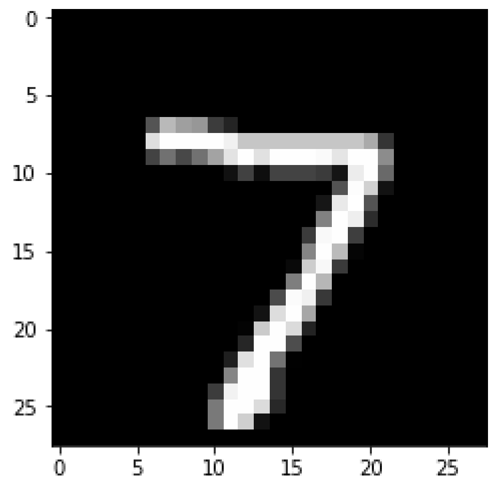
[1]:
```
import keras
from keras import backend as K
from keras.layers import Conv2D, Dense, Dropout, Flatten, MaxPooling2D, Input,␣
↪UpSampling2D
from keras.models import Model
from keras.utils import to_categorical
import matplotlib
%matplotlib inline
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import StratifiedShuffleSplit
from alibi.confidence import TrustScore
```

```
Using TensorFlow backend.
```

[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[0])
```

```
x_train shape: (60000, 28, 28) y_train shape: (60000,)
```

```
[2]: <matplotlib.image.AxesImage at 0x7f7fd1e40128>
```



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
```

20.1 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, verbose=0)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 28, 28, 1)         0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 784)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 10)                7850      
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
_________________________________________________________________
```

[6]:
```
<keras.callbacks.History at 0x7f7fd1e13630>
```

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
```

20.2 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)
    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')
```

(continues on next page)

(continued from previous page)

```
    return autoencoder, encoder, decoder
```

[9]:
```
ae, enc, dec = ae_model()
ae.summary()
ae.fit(x_train, x_train, batch_size=128, epochs=8, validation_data=(x_test, x_test),␣
↪verbose=0)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
model_2 (Model)              (None, 4, 4, 4)           1612
_________________________________________________________________
model_3 (Model)              (None, 28, 28, 1)         1757
=================================================================
Total params: 3,369
Trainable params: 3,369
Non-trainable params: 0
_________________________________________________________________
```

[9]:
```
<keras.callbacks.History at 0x7f7fd24cd5c0>
```

20.3 Calculate Trust Scores

Initialize trust scores:

[10]:
```
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:

[11]:
```
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:

[12]:
```
x_test_enc = enc.predict(x_test)
y_pred = sc.predict(x_test)
score, closest_class = ts.score(x_test_enc, y_pred, k=5)
```

```
Reshaping data from (10000, 4, 4, 4) to (10000, 64) so k-d trees can be queried.
```

Let's inspect which predictions have low and high trust scores:

[13]:
```
n = 5
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 = np.argmax(y_pred[idx_min], axis=1), np.argmax(y_pred[idx_max],␣
↪axis=1)
```

(continues on next page)

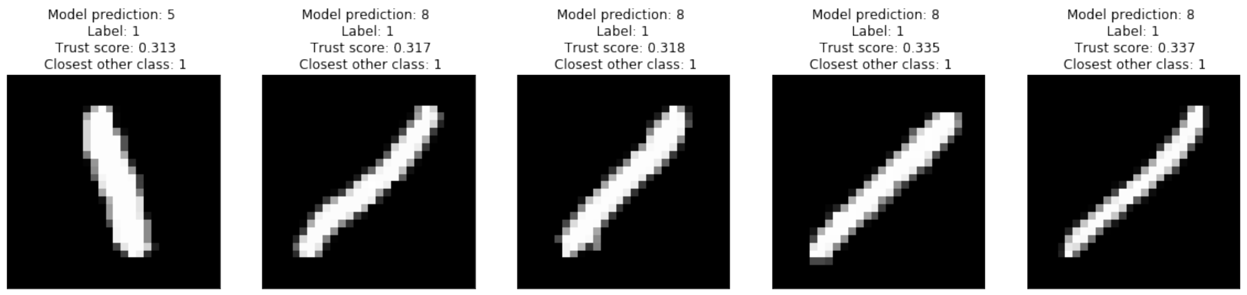
(continued from previous page)

```
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)
```

20.3.1 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.

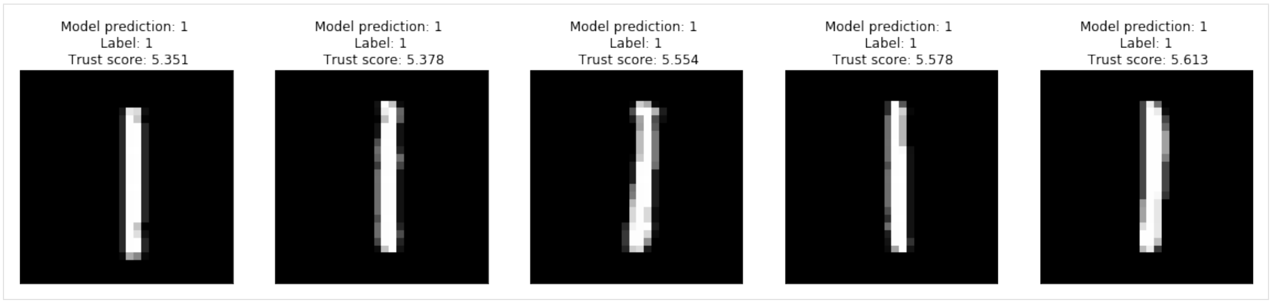
```
[14]: plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(1, n, i+1)
    plt.imshow(imgs_min[i].reshape(28, 28))
    plt.title('Model prediction: {} \n Label: {} \n Trust score: {:.3f}' \
              '\n Closest other class: {}'.format(pred_min[i], label_min[i], score_
↪min[i], closest_min[i]))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



20.3.2 High Trust Scores

The high trust scores on the other hand all are very clear 1's:

```
[15]: 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: {} \n Label: {} \n Trust score: {:.3f}'.format(pred_
↪max[i], label_max[i], score_max[i]))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



20.4 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:

```
[16]: 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:

```
[17]: 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:
```

(continues on next page)

(continued from previous page)

```
        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:

[18]:
```
def run_precision_plt(X, y, nfolds, percentiles, run_model, test_size=.2,
                      plt_title="", plt_names=[], predict_correct=True, classes=10):

    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_sc(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):
```

(continues on next page)

(continued from previous page)

```
        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)
```

20.5 Detect correctly classified examples

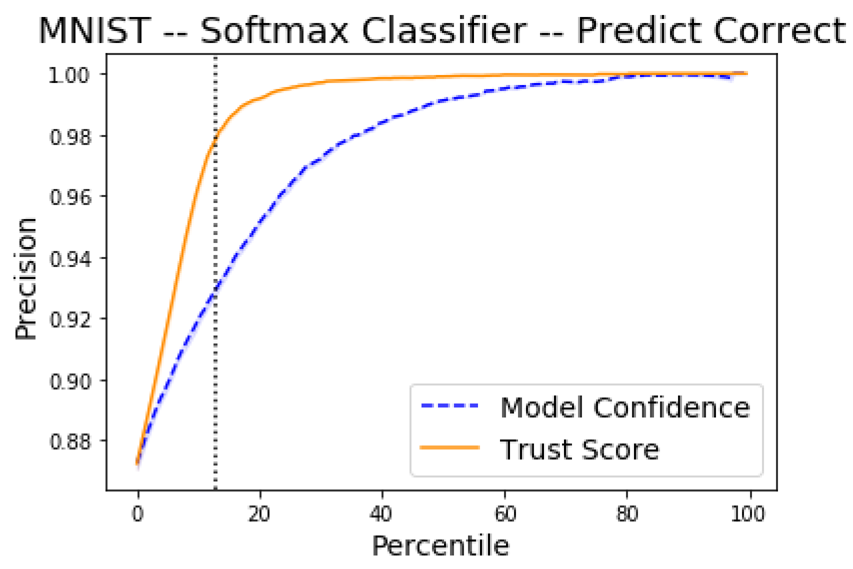
The x-axis on the plot below shows the percentiles for the model prediction probabilities of the predicted class for each instance and for the trust scores. The y-axis represents the precision for each percentile. For each percentile level, we take the test examples whose trust score is above that percentile 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.

```
[19]: 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'
```

```
[20]: 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.
```


MNIST -- Softmax Classifier -- Predict Correct
Precision
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0
20
40
60
80
100
Percentile
Model Confidence
Trust Score

CHAPTER 21

alibi

21.1 alibi package

21.1.1 Subpackages

alibi.confidence package

Submodules

alibi.confidence.trustscore module

alibi.explainers package

Submodules

alibi.explainers.anchor_base module

alibi.explainers.anchor_explanation module

alibi.explainers.anchor_image module

alibi.explainers.anchor_tabular module

alibi.explainers.anchor_text module

alibi.explainers.cem module

alibi.explainers.cfproto module

alibi.explainers.counterfactual module

alibi.outlier package

alibi.utils package

Submodules

alibi.utils.discretizer module

alibi.utils.distance module

alibi.utils.download module

alibi.utils.gradients module

21.1.2 Submodules

alibi.datasets module

alibi.version module

CHAPTER 22

Indices and tables

- genindex
- modindex
- search