MiraiML

Oct 21, 2019

Contents

1 User Guide **3**
1.1 Introduction . . . 3
1.2 MiraiML usability . . . 3
1.3 MiraiML internals . . . 5
1.4 Optimization workflow . . . 7

2 The User's API **9**
2.1 miraiml.SearchSpace . . . 9
2.2 miraiml.Config . . . 10
2.3 miraiml.Engine . . . 12
2.4 miraiml.Status . . . 14
2.5 miraiml.pipeline . . . 15

3 Internal modules' API **19**
3.1 miraiml.core . . . 19
3.2 miraiml.util . . . 23

4 Example notebook **25**
4.1 Building the search spaces . . . 25
4.2 Configuring the Engine . . . 26
4.3 Triggering the Engine . . . 26
4.4 Engine's status analysis . . . 28

Python Module Index **31**

Index **33**

CHAPTER 1

User Guide

1.1 Introduction

Mirai: *future* in japanese.

MiraiML is an asynchronous engine for continuous & autonomous machine learning, built for real-time usage.

- It's asynchronous because it runs in background, allowing you to execute custom Python code as you interact with the engine;
- It's continuous because it can run "forever", always looking for solutions that can achieve better performances;
- It's autonomous because it does not wander on the search space blindly and does not perform exhaustive grid searches. Instead, it combines past attempts to guide its next steps, always allowing itself to jump out of local minima.

MiraiML improves the chosen metric by searching good hyperparameters and sets of features for base statistical models, whilst finding smart ways to combine the predictions of those models in order to achieve even higher scores.

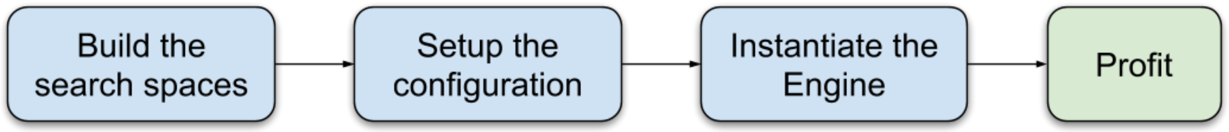
But how can MiraiML help me? And how does it even work?

We're going to address these questions on the next subsections.

1.2 MiraiML usability

Tired of coding the same grid searches, cross-validations, training and predicting scripts over and over? I was. MiraiML does it all with a simple API, so you can spend less time on such mechanical tasks. MiraiML works on the typical train/test scenario, when the data can fit in the RAM. Let's explore the API from a bottom-up perspective.

The basic usage flow is represented in the image below:



1.2.1 Search spaces

MiraiML requires that you define the search spaces in which it will look for solution candidates. In order to instantiate a search space, you need to use the *miraiml.SearchSpace* class. A search space is a combination of an id, a model class and a dictionary of hyperparameters values to be tested. The only requirement is that the model class must implement a `fit` method as well as a `predict` method for regression problems or a `predict_proba` for classification problems. For instance, you can use scikit-learn's models:

```
>>> from sklearn.linear_model import LinearRegression
>>> from miraiml import SearchSpace

>>> search_space = SearchSpace(
...     id = 'Linear Regression',
...     model_class = LinearRegression,
...     parameters_values = dict(
...         fit_intercept = [True, False],
...         normalize = [True, False]
...     )
... )
```

miraiml.SearchSpace also allows you to provide a *parameters_rules* function to deal with prohibitive combinations of hyperparameters. Please refer to its documentation for further understanding.

After you've defined your search spaces, the next step is building the configuration object.

1.2.2 Configuration

The configuration for MiraiML's Engine is defined by an instance of the *miraiml.Config* class, which tells the Engine where to save its local files (checkpoints), the problem type, the function to score the candidate solutions, the search spaces that should be used and a few other things. For instance:

```
>>> from sklearn.metrics import r2_score
>>> from miraiml import Config

>>> config = Config(
...     local_dir = 'miraiml_local',
...     problem_type = 'regression',
...     score_function = r2_score,
...     search_spaces = [search_space]
... )
```

Alright, now we're all set to use the Engine.

1.2.3 The Engine

miraiml.Engine provides a straightforward interface to access its functionalities. The instantiation only requires a configuration object:

```
>>> from miraiml import Engine

>>> engine = Engine(config)
```

Note: You can also provide a `on_improvement` function that will be executed everytime the engine finds a better modeling solution. Check out the API documentation for more information.

Let's use scikit-learn's classic *California Housing* dataset as an example:

```
>>> from sklearn.datasets import fetch_california_housing
>>> import pandas as pd

>>> X, y = fetch_california_housing(return_X_y=True)
>>> data = pd.DataFrame(X)
>>> data['target'] = y

>>> engine.load_train_data(train_data=data, target_column='target')
```

After the training data is loaded, you can trigger the optimization process with:

```
>>> engine.restart()
```

And to interrupt it:

```
>>> engine.interrupt()
```

The *`miraiml.Engine`* documentation contains the full set of functionalities that are available for you.

1.3 MiraiML internals

MiraiML works in cycles. In each cycle, the Engine tries to find better solutions for each search space and for the ensemble. There are three main concepts at play here:

- *Base models* represent solutions in the search space
- *Mirai Seeker* manages the walk through the search spaces
- *Ensembler* attempts weighted combinations of base models

1.3.1 Base models

Fit, predict and validate with a single button.

Base models are the fundamental bricks of the optimization process. A base model is a combination of a model class, a set of parameters and a set of features.

Base models implement a versatile method for predictions, which return predictions for the training data and for the testing data, as well as the score achieved on the training data.

The mechanics of this process is similar to a cross-validation, with a slight difference: the final score is not the mean score of each fold. Instead, the array of predictions is built iteratively and then fully compared to the target column. More precisely:

1. Filter training and testing features

2. Split the training data in N folds
3. **For each fold:**
    - Train the model on the bigger part
    - Make predictions for the smaller part
    - Make predictions for the testing dataset
4. Compute the score for the entire column of predictions on the training dataset
5. Compute the average of the predictions for the testing dataset

Averaging the predictions for the testing dataset may result in slightly better accuracies than expected.

Pipelines

Pipelines can be used as base models when you want to test various ways of pre-processing your data before fitting it with an estimator.

If that's your case, please check out the `miraiml.pipeline` module.

1.3.2 Mirai Seeker

There can be too many base models in the search space and we may not be able to afford exhausive searches. Thus, a smart strategy to search good base models is mandatory.

The engine registers optimization attempts on dataframes called *histories*. These dataframes have columns for each hyperparameter and each feature, as well as a column for the reported scores. The values of the hyperparameters' columns are the values of the hyperparameters themselves. The values of the features' columns are either 0 or 1, which indicate whether the features were used or not. An example of history dataframe for a K-NN classifier with three registries would be:

Hyperparameters		Features		—
n_neighbors	weights	age	gender	score
3	'uniform'	1	0	0.82
2	'distance'	0	1	0.76
4	'uniform'	1	1	0.84

As the history grows, it can be used to generate potentially good base models for future optimization attempts. Currently, the available strategies to create base models are:

- **Random** Generates a completely random base model.
- **Naive** The naive strategy iterates over the history columns (except the score) and groups the data by the current column values using the *mean* aggregation function on the score column. Each value present on the current column can be chosen with a probability that is proportional to the mean score from the *group by* aggregation.

  For instance, if we aggregate the history dataframe above by the column *age*, the mean score of attempts in which the feature *age* was chosen is 0.83 and the mean score of the attempts in which the feature *age* was **not** chosen is 0.76. Now, we choose to use *age* on the next base model with a probability that's proportional to 0.83 and we choose **not to** with a probability that's proportional to 0.76.

  It's called *Naive* because it assumes the strong hypothesis that the columns of history dataframes affect the score independently.

- **Linear Regression** Uses a simple linear regression to model the score as a function of the other history columns. Categorical columns are processed with One-Hot-Encoding. This strategy makes *n*/2 guesses and chooses the best guess according to the linear regression model, where *n* is the size of the history dataframe.

The strategy is chosen stochastically according to the following priority rule:

> *The random strategy will be chosen with a probability of 0.5. If it's not, the other strategies will be chosen with equal probabilities.*

1.3.3 Ensembler

It is possible to combine the predictions of various base models in order to reach even higher scores. This process is done by computing a straightforward linear combination of the base models' predictions.

More precisely, suppose we have a set of base models. For each base model i, let tr_i and ts_i be its predictions for the training and testing dataset, respectively. The ensemble of the base models is based on a set of coefficients w (weights), for which we can compute the combined predictions E_{tr} and E_{ts} for the training and testing datasets, respectively, according to the formula:

$$(E_{tr}, E_{ts}) = \left(\frac{\sum w_i tr_i}{\sum w_i}, \frac{\sum w_i ts_i}{\sum w_i}\right)$$

With a smart choice of w, the score for E_{tr} may be better than the score of any tr_i.

Now, the obvious question is: how to find a good w? This is where the concept of *ensembling cycles* comes into play.

An ensembling cycle is an attempt to generate good weights stochastically, based on the the score of each base model individually. This is done by using triangular distributions.

The weight of the best base model is drawn from the triangular distribution that varies from 0 to 1, with mode 1.

For every other base model i (not a base model with the highest score), the weight is drawn from a triangular distribution that varies from 0 to *range*, with mode 0. It means that its weight will most likely be close to 0 and its upperbound is defined by the *range* variable.

The value of *range* should depend on the relative score of the base model. But preventing it from reaching 1 would be too prohibitive. A solution for this is: *range* is chosen from a triangular distribution that varies from 0 to 1, with mode *normalized*. The variable *normalized* measures the relative quality of the base model.

The value of *normalized* is computed by the formula $(s_i - s_{\text{min}})/(s_{\text{max}} - s_{\text{min}})$, where s_i is the score of the base model and s_{min} and s_{max} are the scores of the worst and the best base models, respectively.

In the end, bad base models can still influence the ensemble, but their probabilities of having high weights are relatively low.

The number of ensembling cycles depend on the time consumed by the other models. The current rule is:

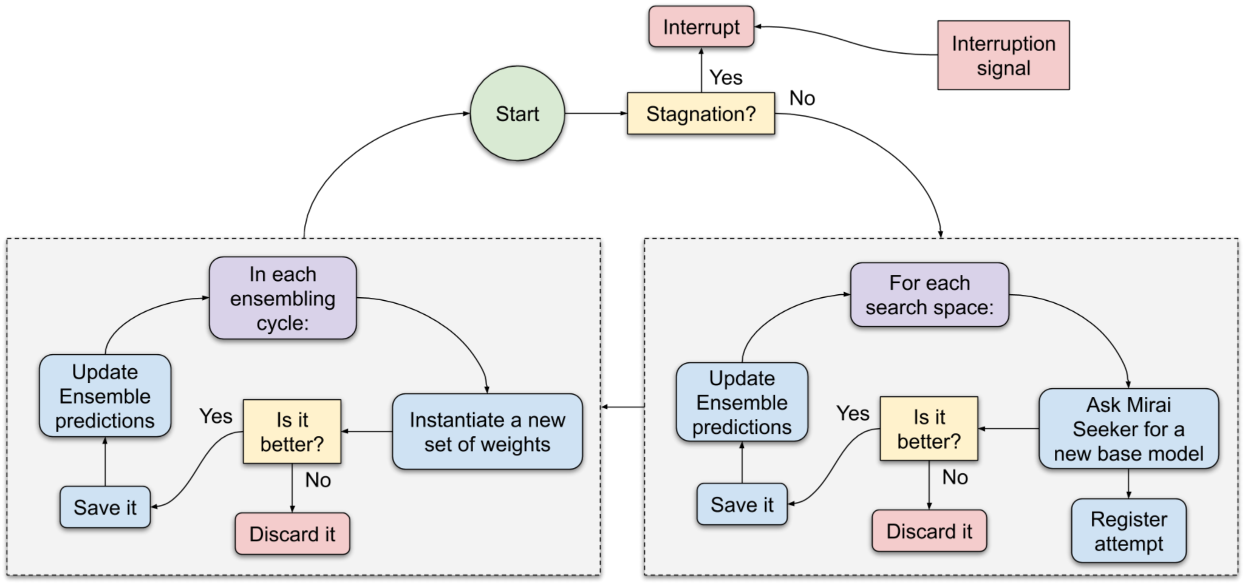
> *The time consumed by the ensemble is limited by the total time consumed by all base models, on average.*

Warning: The score of the Ensemble *may* decrease when the engine finds a better base model and updates its predictions.

1.4 Optimization workflow

The optimization cycle starts by looking for better base models for each search space. Mirai Seeker is responsible for keeping track of old base models attempts in order to provide good guesses for new attempts. If a better base model is found for some search space, the ensembler output is updated with the new predictions. Then, after a new solution is attempted for each search space, the Engine executes the ensembling cycles, looking for better ensembling weights.

Wrapping it all up, the following diagram represents the workflow within an optimization loop:



CHAPTER 2

The User's API

miraiml provides the following components:

- *miraiml.SearchSpace* represents the search space for a base model
- *miraiml.Config* defines the general behavior for *miraiml.Engine*
- *miraiml.Engine* manages the optimization process
- *miraiml.pipeline* has some features related to pipelines **(hot!)**

2.1 miraiml.SearchSpace

class miraiml.**SearchSpace**(*id*, *model_class*, *parameters_values=None*, *parameters_rules=<function SearchSpace.<lambda>>*)

This class represents the search space of hyperparameters for a base model.

Parameters

- **id** (*str*) – The id that will be associated with the models generated within this search space.
- **model_class** (*type*) – Any class that represents a statistical model. It must implement the methods `fit` as well as `predict` for regression or `predict_proba` for classification problems.
- **parameters_values** (*dict, optional, default=None*) – A dictionary containing lists of values to be tested as parameters when instantiating objects of `model_class` for `id`.
- **parameters_rules** (*function, optional, default=lambda x: None*) – A function that constrains certain parameters because of the values assumed by others. It must receive a dictionary as input and doesn't need to return anything. Not used if `parameters_values` has no keys.

Warning: Make sure that the parameters accessed in `parameters_rules` exist in the set of parameters defined on `parameters_values`, otherwise the engine will attempt to access an invalid key.

Raises `NotImplementedError` if a model class does not implement `fit` or none of `predict` or `predict_proba`.

Raises `TypeError` if some parameter is of a prohibited type.

Raises `ValueError` if a provided `id` is not allowed.

Example

```
>>> import numpy as np
>>> from sklearn.linear_model import LogisticRegression
>>> from miraiml import SearchSpace

>>> def logistic_regression_parameters_rules(parameters):
...     if parameters['solver'] in ['newton-cg', 'sag', 'lbfgs']:
...         parameters['penalty'] = 'l2'

>>> search_space = SearchSpace(
...     id = 'Logistic Regression',
...     model_class = LogisticRegression,
...     parameters_values = {
...         'penalty': ['l1', 'l2'],
...         'C': np.arange(0.1, 2, 0.1),
...         'max_iter': np.arange(50, 300),
...         'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'],
...         'random_state': [0]
...     },
...     parameters_rules = logistic_regression_parameters_rules
... )
```

Warning: **Do not** allow `random_state` assume multiple values. If `model_class` has a `random_state` parameter, force the engine to always choose the same value by providing a list with a single element.

Allowing `random_state` to assume multiple values will confuse the engine because the scores will be unstable even with the same choice of hyperparameters and features.

2.2 miraiml.Config

class miraiml.**Config**(*local_dir*, *problem_type*, *score_function*, *search_spaces*, *use_all_features=False*, *n_folds=5*, *stratified=True*, *ensemble_id=None*, *stagnation=60*)

This class defines the general behavior of the engine.

Parameters

- **local_dir** (*str*) – The name of the folder in which the engine will save its internal files. If the directory doesn't exist, it will be created automatically. `..` and `/` are not allowed to compose `local_dir`.

- **problem_type** (*str*) – 'classification' or 'regression'. The problem type. Multi-class classification problems are not supported.
- **search_spaces** (*list*) – The list of *miraiml.SearchSpace* objects to optimize. If search_spaces has length 1, the engine will not run ensemble cycles.
- **score_function** (*function*) – A function that receives the "truth" and the predictions (in this order) and returns the score. Bigger scores must mean better models.
- **use_all_features** (*bool, optional, default=False*) – Whether to force MiraiML to always use all features or not.
- **n_folds** (*int, optional, default=5*) – The number of folds for the fitting/predicting process. The minimum value allowed is 2.
- **stratified** (*bool, optional, default=True*) – Whether to stratify folds on target or not. Only used if problem_type == 'classification'.
- **ensemble_id** (*str, optional, default=None*) – The id for the ensemble. If none is given, the engine will not ensemble base models.
- **stagnation** (*int or float, optional, default=60*) – The amount of time (in minutes) for the engine to automatically interrupt itself if no improvement happens. Negative numbers are interpreted as "infinite".

Warning: Stagnation checks only happen after the engine finishes at least one optimization cycle. In other words, every base model and the ensemble (if set) must be scored at least once.

Raises NotImplementedError if a model class does not implement the proper method for prediction.

Raises TypeError if some parameter is not of its allowed type.

Raises ValueError if some parameter has an invalid value.

Example

```
>>> from sklearn.metrics import roc_auc_score
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.tree import DecisionTreeClassifier
>>> from miraiml import SearchSpace, Config

>>> search_spaces = [
...     SearchSpace('Naive Bayes', GaussianNB),
...     SearchSpace('Decicion Tree', DecisionTreeClassifier)
... ]

>>> config = Config(
...     local_dir = 'miraiml_local',
...     problem_type = 'classification',
...     score_function = roc_auc_score,
...     search_spaces = search_spaces,
...     use_all_features = False,
...     n_folds = 5,
...     stratified = True,
...     ensemble_id = 'Ensemble',
...     stagnation = -1
... )
```

2.3 miraiml.Engine

class miraiml.**Engine**(*config*, *on_improvement=None*)

This class offers the controls for the engine.

Parameters

- **config** (miraiml.Config) – The configurations for the behavior of the engine.
- **on_improvement** (*function, optional, default=None*) – A function that will be executed everytime the engine finds an improvement for some id. It must receive a status parameter, which is the return of the method *request_status()* (an instance of *miraiml.Status*).

Raises TypeError if config is not an instance of *miraiml.Config* or on_improvement (if provided) is not callable.

Example

```
>>> from sklearn.metrics import roc_auc_score
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.tree import DecisionTreeClassifier
>>> from miraiml import SearchSpace, Config, Engine

>>> search_spaces = [
...     SearchSpace('Naive Bayes', GaussianNB),
...     SearchSpace('Decision Tree', DecisionTreeClassifier)
... ]

>>> config = Config(
...     local_dir = 'miraiml_local',
...     problem_type = 'classification',
...     score_function = roc_auc_score,
...     search_spaces = search_spaces,
...     ensemble_id = 'Ensemble'
... )

>>> def on_improvement(status):
...     print('Scores:', status.scores)

>>> engine = Engine(config, on_improvement=on_improvement)
```

is_running	Tells whether the engine is running or not.
interrupt	Makes the engine stop on the first opportunity.
load_train_data	Interrupts the engine and loads the train dataset.
load_test_data	Interrupts the engine and loads the test dataset.
shuffle_train_data	Interrupts the engine and shuffles the training data.
reconfigure	Interrupts the engine and loads a new configuration.
restart	Interrupts the engine and starts again from last checkpoint (if any).
request_status	Queries the current status of the engine.

is_running()

Tells whether the engine is running or not.

Return type bool

Returns `True` if the engine is running and `False` otherwise.

`interrupt`()

Makes the engine stop on the first opportunity.

Note: This method is **not** asynchronous. It will wait until the engine stops.

`load_train_data`(*train_data*, *target_column*, *restart=False*)

Interrupts the engine and loads the train dataset. All of its columns must be either instances of `str` or `int`.

Warning: Loading new training data will **always** trigger the loss of history for optimization.

Parameters

- **`train_data`** (*`pandas.DataFrame`*) – The training data.
- **`target_column`** (*`str or int`*) – The target column identifier.
- **`restart`** (*`bool, optional, default=False`*) – Whether to restart the engine after updating data or not.

Raises `TypeError` if `train_data` is not an instance of `pandas.DataFrame`.

Raises `ValueError` if `target_column` is not a column of `train_data` or if some column name is of a prohibited type.

`load_test_data`(*test_data*, *restart=False*)

Interrupts the engine and loads the test dataset. All of its columns must be columns in the train data.

The test dataset is the one for which we don't have the values for the target column. This method should be used to load data in production.

Warning: This method can only be called after *`miraiml.Engine.load_train_data()`*

Parameters

- **`test_data`** (*`pandas.DataFrame, optional, default=None`*) – The testing data. Use the default value if you don't need to make predictions for data with unknown labels.
- **`restart`** (*`bool, optional, default=False`*) – Whether to restart the engine after loading data or not.

Raises `RuntimeError` if this method is called before loading the train data.

Raises `ValueError` if the column names are not consistent.

`clean_test_data`(*restart=False*)

Cleans the test data from the buffer.

Note: Keep in mind that if you don't intend to make predictions for unlabeled data, the engine will run faster with a clean test data buffer.

Parameters **`restart`** (*`bool, optional, default=False`*) – Whether to restart the engine after cleaning test data or not.

`shuffle_train_data`(*restart=False*)
Interrupts the engine and shuffles the training data.

Parameters **`restart`** (*`bool, optional, default=False`*) – Whether to restart the engine after shuffling data or not.

Raises `RuntimeError` if the engine has no data loaded.

Note: It's a good practice to shuffle the training data periodically to avoid overfitting on a particular folding pattern.

`reconfigure`(*config*, *restart=False*)
Interrupts the engine and loads a new configuration.

Warning: Reconfiguring the engine will **always** trigger the loss of history for optimization.

Parameters

- **`config`** (`miraiml.Config`) – The configurations for the behavior of the engine.
- **`restart`** (*`bool, optional, default=False`*) – Whether to restart the engine after reconfiguring it or not.

`restart`()
Interrupts the engine and starts again from last checkpoint (if any). It is also used to start the engine for the first time.

Raises `RuntimeError` if no data is loaded.

`request_status`()
Queries the current status of the engine.

Return type *miraiml.Status*

Returns The current status of the engine in the form of a dictionary. If no score has been computed yet, returns `None`.

2.4 miraiml.Status

`class` `miraiml.`**`Status`**(***kwargs*)
Represents the current status of the engine. Objects of this class are not supposed to be instantiated by the user. Rather, they are returned by the *`miraiml.Engine.request_status()`* method.

The following attributes are accessible:

- `best_id`: the id of the best base model (or ensemble)
- `scores`: a dictionary containing the current score of each id
- `train_predictions`: a `pandas.DataFrame` object containing the predictions for the train data for each id

- `test_predictions`: a `pandas.DataFrame` object containing the predictions for the test data for each id
- `ensemble_weights`: a dictionary containing the ensemble weights for each base model id
- `base_models`: a dictionary containing the characteristics of each base model (accessed by its respective id)
- `histories`: a dictionary of `pandas.DataFrame` objects for each id, containing the history of base models attempts and their respective scores. Hyperparameters columns end with the `'__(hyperparameter)'` suffix and features columns end with the `'__(feature)'` suffix. The score column can be accessed with the key `'score'`. For more information, please check the *User Guide*.

The characteristics of each base model are represent by dictionaries, containing the following keys:

- `'model_class'`: The name of the base model's modeling class
- `'parameters'`: The dictionary of hyperparameters values
- `'features'`: The list of features used

`build_report`(*include_features=False*)

Returns the report of the current status of the engine in a formatted string.

Parameters **include_features** (*bool, optional, default=False*) – Whether to include the list of features on the report or not (may cause some visual mess).

Return type str

Returns The formatted report.

2.5 miraiml.pipeline

`miraiml.pipeline` contains a function that lets you build your own pipeline classes. It also contains a few pre-defined pipelines for baselines.

`compose`	A function that defines pipeline classes dinamically.
`NaiveBayesBaseliner`	This is a baseline pipeline for classification problems.
`LinearRegressionBaseliner`	This is a baseline pipeline for regression problems.

`miraiml.pipeline.`**`compose`**(*steps*)

A function that defines pipeline classes dinamically. It builds a pipeline class that can be instantiated with particular parameters for each of its transformers/estimator without needing to call `set_params` as you would do with scikit-learn's Pipeline when performing hyperparameters optimizations.

Similarly to scikit-learn's Pipeline, `steps` is a list of tuples containing an alias and the respective pipeline element. Although, since this function is a class factory, you shouldn't instantiate the transformer/estimator as you would do with scikit-learn's Pipeline. Thus, this is how *`compose()`* should be called:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.preprocessing import StandardScaler

>>> from miraiml.pipeline import compose

>>> MyPipelineClass = compose(
...     steps = [
...         ('scaler', StandardScaler), # StandardScaler instead of␣
↪StandardScaler()
```

(continues on next page)

(continued from previous page)

```
...         ('rfc', RandomForestClassifier) # No instantiation either
...     ]
... )
```

And then, in order to instantiate `MyPipelineClass` with the desired parameters, you just need to refer to them as a concatenation of their respective class aliases and their names, separated by `'__'`.

```
>>> pipeline = MyPipelineClass(scaler__with_mean=False, rfc__max_depth=3)
```

If you want to know which parameters you're allowed to play with, just call `get_params`:

```
>>> params = pipeline.get_params()
>>> print("\n".join(params))
scaler__with_mean
scaler__with_std
rfc__bootstrap
rfc__class_weight
rfc__criterion
rfc__max_depth
rfc__max_features
rfc__max_leaf_nodes
rfc__min_impurity_decrease
rfc__min_impurity_split
rfc__min_samples_leaf
rfc__min_samples_split
rfc__min_weight_fraction_leaf
rfc__n_estimators
rfc__n_jobs
rfc__oob_score
rfc__random_state
rfc__verbose
rfc__warm_start
```

You can check the available methods for your instantiated pipelines on the documentation for *`miraiml.core.BasePipelineClass`*, which is the class from which the composed classes inherit from.

The intended purpose of such pipeline classes is that they can work as base models to build instances of *`miraiml.SearchSpace`*.

```
>>> from miraiml import SearchSpace

>>> search_space = SearchSpace(
...     id='MyPipelineClass',
...     model_class=MyPipelineClass,
...     parameters_values=dict(
...         scaler__with_mean=[True, False],
...         scaler__with_std=[True, False],
...         rfc__max_depth=[3, 4, 5, 6]
...     )
... )
```

Parameters **`steps`** (*`list`*) – The list of pairs (alias, class) to define the pipeline.

Warning: Repeated aliases are not allowed and none of the aliases can start with numbers or contain `'__'`.

> The classes used to compose a pipeline **must** implement `get_params` and `set_params`, such as scikit-learn's classes, or `compose()` **will break**.

Return type type

Returns The composed pipeline class

Raises `TypeError` if an alias is not a string.

Raises `ValueError` if an alias has an invalid name.

Raises `NotImplementedError` if some class of the pipeline does not implement the required methods.

class `miraiml.pipeline.`**`NaiveBayesBaseliner`**

This is a baseline pipeline for classification problems. It's composed by the following transformers/estimator:

1. `sklearn.preprocessing.OneHotEncoder`
2. `sklearn.impute.SimpleImputer`
3. `sklearn.preprocessing.MinMaxScaler`
4. `sklearn.naive_bayes.GaussianNB`

The available parameters to tweak are:

```
>>> from miraiml.pipeline import NaiveBayesBaseliner

>>> for param in NaiveBayesBaseliner().get_params():
...     print(param)
...
ohe__categorical_features
ohe__categories
ohe__drop
ohe__dtype
ohe__handle_unknown
ohe__n_values
ohe__sparse
impute__add_indicator
impute__fill_value
impute__missing_values
impute__strategy
impute__verbose
min_max__feature_range
naive__priors
naive__var_smoothing
```

class `miraiml.pipeline.`**`LinearRegressionBaseliner`**

This is a baseline pipeline for regression problems. It's composed by the following transformers/estimator:

1. `sklearn.preprocessing.OneHotEncoder`
2. `sklearn.impute.SimpleImputer`
3. `sklearn.preprocessing.MinMaxScaler`
4. `sklearn.linear_model.LinearRegression`

The available parameters to tweak are:

```
>>> from miraiml.pipeline import LinearRegressionBaseliner

>>> for param in LinearRegressionBaseliner().get_params():
...     print(param)
...
ohe__categorical_features
ohe__categories
ohe__drop
ohe__dtype
ohe__handle_unknown
ohe__n_values
ohe__sparse
impute__add_indicator
impute__fill_value
impute__missing_values
impute__strategy
impute__verbose
min_max__feature_range
lin_reg__fit_intercept
lin_reg__n_jobs
lin_reg__normalize
```

CHAPTER 3

Internal modules' API

The documentation related to these modules is meant for developers.

3.1 miraiml.core

`miraiml.core` contains internal classes responsible for the optimization process.

class `miraiml.core.`**`BaseModel`**(*model_class*, *parameters*, *features*)

Represents an element from the search space, defined by an instance of `miraiml.SearchSpace` and a set of features.

Read more in the *User Guide*.

Parameters

- **`model_class`** (*`type`*) – A statistical model class that must implement the methods `fit` and `predict` for regression or `predict_proba` classification problems.
- **`parameters`** (*`dict`*) – The parameters that will be used to instantiate objects of `model_class`.
- **`features`** (*`list`*) – The list of features that will be used to train the statistical model.

`predict`(*X_train*, *y_train*, *X_test*, *config*)

Performs the predictions for the training and testing datasets and also computes the score of the model.

Parameters

- **`X_train`** (*`pandas.DataFrame`*) – The dataframe that contains the training inputs for the model.
- **`y_train`** (*`pandas.Series or numpy.ndarray`*) – The training targets for the model.
- **`X_test`** (*`pandas.DataFrame`*) – The dataframe that contains the testing inputs for the model.

- **config** (`miraiml.Config`) – The configuration of the engine.

Return type tuple

Returns

`(train_predictions, test_predictions, score)`

- `train_predictions`: The predictions for the training dataset
- `test_predictions`: The predictions for the testing dataset
- `score`: The score of the model on the training dataset

Raises `RuntimeError` when fitting or predicting doesn't work.

`miraiml.core.`**`dump_base_model`**(*base_model*, *path*)

Saves the characteristics of a base model as a checkpoint.

Parameters

- **base_model** (`miraiml.core.BaseModel`) – The base model to be saved
- **path** (*str*) – The path to save the base model

Return type tuple

Returns `(train_predictions, test_predictions, score)`

`miraiml.core.`**`load_base_model`**(*model_class*, *path*)

Loads the characteristics of a base model from disk and returns its respective instance of *`miraiml.core.BaseModel`*.

Parameters

- **model_class** (*type*) – The model class related to the base model
- **path** (*str*) – The path to load the base model from

Return type *miraiml.core.BaseModel*

Returns The base model loaded from disk

class `miraiml.core.`**`MiraiSeeker`**(*search_spaces*, *all_features*, *config*)

This class implements a smarter way of searching good parameters and sets of features.

Read more in the *User Guide*.

Parameters

- **base_models_ids** (*list*) – The list of base models' ids to keep track of.
- **all_features** (*list*) – A list containing all available features.
- **config** (`miraiml.Config`) – The configuration of the engine.

`reset`()

Deletes all base models registries.

`parameters_features_to_dataframe`(*parameters*, *features*, *score*)

Creates an entry for a history.

Parameters

- **parameters** (*list*) – The set of parameters to transform.
- **parameters** – The set of features to transform.
- **score** (*float*) – The score to transform.

register_base_model(*id*, *base_model*, *score*)
Registers the performance of a base model and its characteristics.

Parameters

- **id** (*str*) – The id associated with the base model.
- **base_model** (`miraiml.core.BaseModel`) – The base model being registered.
- **score** (*float*) – The score of `base_model`.

is_ready(*id*)
Tells whether the history of an id is large enough for more advanced strategies or not.

Parameters **id** (*str*) – The id to be inspected.

Return type bool

Returns Whether `id` can be used to generate parameters and features lists or not.

seek(*id*)
Manages the search strategy for better solutions.

With a probability of 0.5, the random strategy will be chosen. If it's not, the other strategies will be chosen with equal probabilities.

Parameters **id** (*str*) – The id for which a new base model is required.

Return type *miraiml.core.BaseModel*

Returns The next base model for exploration.

Raises `KeyError` if `parameters_rules` tries to access an invalid key.

random_search(*id*)
Generates completely random sets of parameters and features.

Parameters **all_features** (*list*) – The list of available features.

Return type tuple

Returns `(parameters, features)` Respectively, the dictionary of parameters and the list of features that can be used to generate a new base model.

naive_search(*id*)
Characteristics that achieved higher scores have independently higher chances of being chosen again.

Parameters **id** (*str*) – The id for which we want a new set of parameters and features.

Return type tuple

Returns `(parameters, features)` Respectively, the dictionary of parameters and the list of features that can be used to generate a new base model.

linear_regression_search(*id*)
Uses the history to model the score with a linear regression. Guesses the scores of $n/2$ random sets of parameters and features, where n is the size of the history. The one with the highest score is chosen.

Parameters **id** (*str*) – The id for which we want a new set of parameters and features.

Return type tuple

Returns `(parameters, features)` Respectively, the dictionary of parameters and the list of features that can be used to generate a new base model.

class `miraiml.core.`**`Ensembler`**(*base_models_ids*, *y_train*, *train_predictions_df*, *test_predictions_df*, *scores*, *config*)

Performs the ensemble of the base models and optimizes its weights.

Read more in the *User Guide*.

Parameters

- **`y_train`** (*`pandas.Series or numpy.ndarray`*) – The target column.
- **`base_models_ids`** (*`list`*) – The list of base models' ids to keep track of.
- **`train_predictions_df`** (*`pandas.DataFrame`*) – The dataframe of predictions for the training dataset.
- **`test_predictions_df`** (*`pandas.DataFrame`*) – The dataframe of predictions for the testing dataset.
- **`scores`** (*`dict`*) – The dictionary of scores.
- **`config`** (`miraiml.Config`) – The configuration of the engine.

`interrupt`()

Sets an internal flag to interrupt the optimization process on the first opportunity.

`update`()

Updates the ensemble with the newest predictions from the base models.

`gen_weights`()

Generates the ensemble weights according to the score of each base model. Higher scores have higher chances of generating higher weights.

Return type dict

Returns A dictionary containing the weights for each base model id.

`ensemble`(*weights*)

Performs the ensemble of the current predictions of each base model.

Parameters **`weights`** (*`dict`*) – A dictionary containing the weights related to the id of each base model.

Return type tuple

Returns

`(train_predictions, test_predictions, score)`

- `train_predictions`: The ensemble predictions for the training dataset
- `test_predictions`: The ensemble predictions for the testing dataset
- `score`: The score of the ensemble on the training dataset

`optimize`(*max_duration*)

Performs ensembling cycles for `max_duration` seconds.

Parameters **`max_duration`** (*`float`*) – The maximum duration allowed for the optimization process.

Return type bool

Returns `True` if a better set of weights was found and `False` otherwise.

class `miraiml.core.`**`BasePipelineClass`**(***params*)

This is the base class for your custom pipeline classes.

Warning: Instantiating this class directly **does not work**.

`get_params``()`

Gets the list of parameters that can be set.

Parameters **`X`** (`iterable`) – Data to predict on.

Return type list

Returns The list of allowed parameters

`set_params``(`***params*`)`

Sets the parameters for the pipeline. You can check the parameters that are allowed to be set by calling *`get_params()`*.

Return type *miraiml.core.BasePipelineClass*

Returns self

`fit``(`*X*, *y*`)`

Fits the pipeline to `X` using `y` as the target.

Parameters

- **`X`** (`iterable`) – The training data.
- **`y`** (`iterable`) – The target.

Return type *miraiml.core.BasePipelineClass*

Returns self

`predict``(`*X*`)`

Predicts the class for each element of `X` in case of classification problems or the estimated target value in case of regression problems.

Parameters **`X`** (`iterable`) – Data to predict on.

Return type numpy.ndarray

Returns The set of predictions

`predict_proba``(`*X*`)`

Returns the probabilities for each class. Available only if your end estimator implements it.

Parameters **`X`** (`iterable`) – Data to predict on.

Return type numpy.ndarray

Returns The probabilities for each class

3.2 miraiml.util

`miraiml.util` provides utility functions that are used by higher level modules.

`miraiml.util.`**`load`**`(`*path*`)`

A clean *pickle.load* wrapper for binary files.

Parameters **`path`** (`string`) – The path of the binary file to be loaded.

Return type object

Returns The loaded object.

`miraiml.util.`**`dump`**(*obj*, *path*)

Optimizes the process of writing objects on disc by triggering a thread.

Parameters

- **`obj`** (*`object`*) – The object to be dumped to the binary file.
- **`path`** (*`string`*) – The path of the binary file to be written.

`miraiml.util.`**`sample_random_len`**(*lst*)

Returns a sample of random size from the list `lst`. The minimum length of the returned list is 1.

Parameters **`lst`** (*`list`*) – A list containing the elements to be sampled.

Return type sampled_lst: list

Returns The randomly sampled elements from `lst`.

`miraiml.util.`**`is_valid_filename`**(*filename*)

Tells whether a string can be used as a safe file name or not.

Parameters **`filename`** (*`str`*) – The file name.

Return type bool

Returns Whether `filename` is a valid file name or not.

`miraiml.util.`**`is_valid_pipeline_name`**(*pipeline_name*)

Tells whether a string can be used to compose pipelines or not.

Parameters **`pipeline_name`** (*`str`*) – The file name.

Return type bool

Returns Whether `pipeline_name` is a valid name or not.

CHAPTER 4

Example notebook

This notebook will cover a regression case using scikit-learn's *California Housing* dataset.

```
from sklearn.datasets import fetch_california_housing
import pandas as pd

X, y = fetch_california_housing(data_home='miraiml_local', return_X_y=True)
data = pd.DataFrame(X)
data['target'] = y
```

```
Downloading Cal. housing from https://ndownloader.figshare.com/files/5976036 to␣
↪miraiml_local
```

Let's split the data into training and testing data. In a real case scenario, we'd only have labels for training data.

```
from sklearn.model_selection import train_test_split

train_data, test_data = train_test_split(data, test_size=0.2)
```

4.1 Building the search spaces

Let's compare (and ensemble) a `KNeighborsRegressor` and a pipeline composed by `StandardScaler` and a `LinearRegression`.

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler

from miraiml import SearchSpace
from miraiml.pipeline import compose

Pipeline = compose(
```

(continues on next page)

(continued from previous page)

```
    [('scaler', StandardScaler), ('lin_reg', LinearRegression)]
)

search_spaces = [
    SearchSpace(
        id='k-NN Regressor',
        model_class=KNeighborsRegressor,
        parameters_values=dict(
            n_neighbors=range(2, 9),
            weights=['uniform', 'distance'],
            p=range(2, 5)
        )
    ),
    SearchSpace(
        id='Pipeline',
        model_class=Pipeline,
        parameters_values=dict(
            scaler__with_mean=[True, False],
            scaler__with_std=[True, False],
            lin_reg__fit_intercept=[True, False]
        )
    )
]
```

4.2 Configuring the Engine

For this demonstration, let's use `r2_score` to evaluate our modeling.

```
from sklearn.metrics import r2_score

from miraiml import Config

config = Config(
    local_dir='miraiml_local',
    problem_type='regression',
    score_function=r2_score,
    search_spaces=search_spaces,
    ensemble_id='Ensemble'
)
```

4.3 Triggering the Engine

Let's also print the scores everytime the Engine finds a better solution.

```
from miraiml import Engine

def on_improvement(status):
    scores = status.scores
    for key in sorted(scores.keys()):
        print('{}: {}'.format(key, round(scores[key], 3)), end='; ')
    print()
```

(continues on next page)

(continued from previous page)

```

engine = Engine(config=config, on_improvement=on_improvement)
```

Now we're ready to load the data.

```
engine.load_train_data(train_data, 'target')
engine.load_test_data(test_data)
```

Let's leave it running for 2 minutes, shuffle the train data, let it run for 2 more minutes and then interrupt it.

```
from time import sleep

engine.restart()

sleep(120)

print('\nShuffling train data')
engine.shuffle_train_data(restart=True)

sleep(120)

engine.interrupt()
```

```
Ensemble: 0.118; Pipeline: -3.214; k-NN Regressor: 0.118;
Ensemble: 0.142; Pipeline: -3.214; k-NN Regressor: 0.142;
Ensemble: 0.143; Pipeline: 0.467; k-NN Regressor: 0.142;
Ensemble: 0.474; Pipeline: 0.467; k-NN Regressor: 0.142;
Ensemble: 0.473; Pipeline: 0.467; k-NN Regressor: 0.172;
Ensemble: 0.509; Pipeline: 0.503; k-NN Regressor: 0.172;
Ensemble: 0.509; Pipeline: 0.503; k-NN Regressor: 0.172;
Ensemble: 0.525; Pipeline: 0.503; k-NN Regressor: 0.321;
Ensemble: 0.539; Pipeline: 0.503; k-NN Regressor: 0.321;
Ensemble: 0.552; Pipeline: 0.503; k-NN Regressor: 0.521;
Ensemble: 0.566; Pipeline: 0.503; k-NN Regressor: 0.521;
Ensemble: 0.565; Pipeline: 0.503; k-NN Regressor: 0.538;
Ensemble: 0.566; Pipeline: 0.503; k-NN Regressor: 0.538;
Ensemble: 0.566; Pipeline: 0.503; k-NN Regressor: 0.538;
Ensemble: 0.566; Pipeline: 0.512; k-NN Regressor: 0.538;
Ensemble: 0.566; Pipeline: 0.512; k-NN Regressor: 0.538;
Ensemble: 0.566; Pipeline: 0.512; k-NN Regressor: 0.538;
Ensemble: 0.587; Pipeline: 0.512; k-NN Regressor: 0.544;
Ensemble: 0.597; Pipeline: 0.536; k-NN Regressor: 0.544;
Ensemble: 0.598; Pipeline: 0.536; k-NN Regressor: 0.544;
Ensemble: 0.648; Pipeline: 0.536; k-NN Regressor: 0.659;
Ensemble: 0.666; Pipeline: 0.536; k-NN Regressor: 0.659;
Ensemble: 0.68; Pipeline: 0.536; k-NN Regressor: 0.665;
Ensemble: 0.681; Pipeline: 0.536; k-NN Regressor: 0.665;
Ensemble: 0.681; Pipeline: 0.536; k-NN Regressor: 0.665;
Ensemble: 0.681; Pipeline: 0.536; k-NN Regressor: 0.665;
Ensemble: 0.695; Pipeline: 0.584; k-NN Regressor: 0.665;
Ensemble: 0.698; Pipeline: 0.584; k-NN Regressor: 0.665;
Ensemble: 0.698; Pipeline: 0.597; k-NN Regressor: 0.665;
Ensemble: 0.698; Pipeline: 0.597; k-NN Regressor: 0.665;
Ensemble: 0.698; Pipeline: 0.597; k-NN Regressor: 0.665;
Ensemble: 0.698; Pipeline: 0.597; k-NN Regressor: 0.665;
Ensemble: 0.698; Pipeline: 0.597; k-NN Regressor: 0.665;
```

(continues on next page)

(continued from previous page)

```
Ensemble: 0.698; Pipeline: 0.597; k-NN Regressor: 0.665;
Ensemble: 0.698; Pipeline: 0.597; k-NN Regressor: 0.665;
Ensemble: 0.698; Pipeline: 0.597; k-NN Regressor: 0.687;
Ensemble: 0.7; Pipeline: 0.597; k-NN Regressor: 0.687;
Ensemble: 0.7; Pipeline: 0.597; k-NN Regressor: 0.687;
Ensemble: 0.738; Pipeline: 0.597; k-NN Regressor: 0.72;
Ensemble: 0.738; Pipeline: 0.597; k-NN Regressor: 0.72;
Ensemble: 0.738; Pipeline: 0.597; k-NN Regressor: 0.72;
Ensemble: 0.754; Pipeline: 0.597; k-NN Regressor: 0.753;
Ensemble: 0.757; Pipeline: 0.597; k-NN Regressor: 0.753;
Ensemble: 0.757; Pipeline: 0.597; k-NN Regressor: 0.753;
Ensemble: 0.757; Pipeline: 0.597; k-NN Regressor: 0.753;
Ensemble: 0.757; Pipeline: 0.597; k-NN Regressor: 0.753;

Shuffling train data
Ensemble: 0.757; Pipeline: 0.596; k-NN Regressor: 0.753;
Ensemble: 0.757; Pipeline: 0.596; k-NN Regressor: 0.753;
Ensemble: 0.757; Pipeline: 0.596; k-NN Regressor: 0.753;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
Ensemble: 0.758; Pipeline: 0.596; k-NN Regressor: 0.755;
```

4.4 Engine's status analysis

```
status = engine.request_status()
```

Let's see the status report.

```
print(status.build_report(include_features=True))
```

```
########################
best id: Ensemble
best score: 0.7583702712570008
########################
ensemble weights:
    k-NN Regressor: 0.4325346249356786
    Pipeline: 0.06615069839850787
########################
all scores:
    Ensemble: 0.7583702712570008
    k-NN Regressor: 0.7545806614607227
    Pipeline: 0.5963819838101254
########################
id: Pipeline
model class: MiraiPipeline
```

(continues on next page)

(continued from previous page)

```
n features: 8
parameters:
    lin_reg__fit_intercept: True
    scaler__with_mean: False
    scaler__with_std: False
features: 0, 1, 2, 3, 4, 5, 6, 7
########################
id: k-NN Regressor
model class: KNeighborsRegressor
n features: 6
parameters:
    n_neighbors: 6
    p: 2
    weights: distance
features: 0, 2, 3, 5, 6, 7
```

4.4.1 k-NN Regressor's history

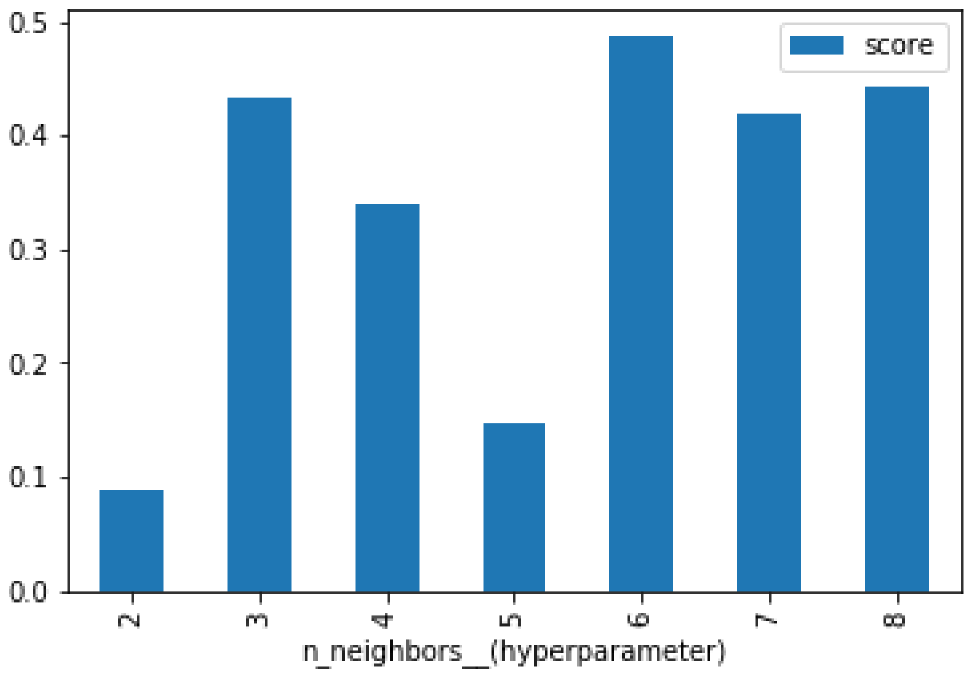
How does the k-NN Regressor's scores change with `n_neighbors`, on average?

```
import matplotlib.pyplot as plt
%matplotlib inline

knn_history = status.histories['k-NN Regressor']

knn_history[['n_neighbors__(hyperparameter)', 'score']]\
    .groupby('n_neighbors__(hyperparameter)').mean()\
    .plot.bar()

plt.show()
```

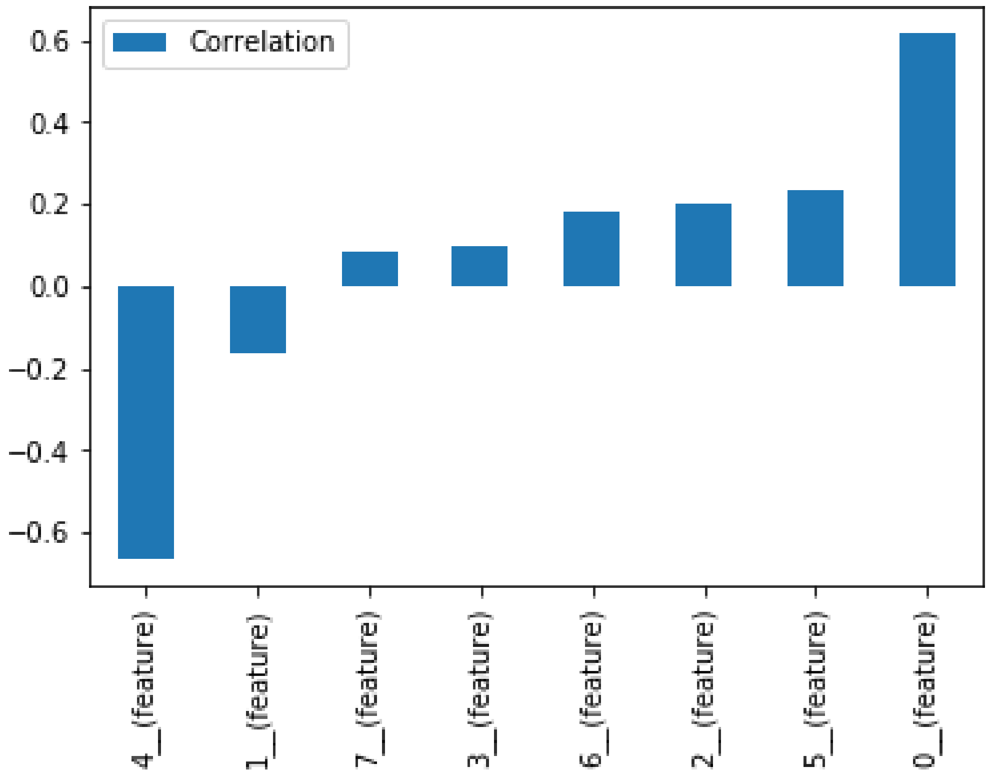


png

We can also see how the presence of features (0 or 1) correlate with the score. These results can work as some sort of feature importance.

```
knn_history[[col for col in knn_history if col.endswith('(feature)')] + ['score']]\
    .corr()['score'][:-1]\
    .sort_values()\
    .plot.bar(label='Correlation')

plt.legend()
plt.show()
```



png

4.4.2 Theoretical performance in production

Again, in practice we wouldn't have labels for `test_data`. But since we do have labels here, how would MiraiML perform on the test dataset?

```
r2_score(test_data['target'], status.test_predictions['Ensemble'])
```

```
0.7802410717298023
```

Python Module Index

m

miraiml, 9
miraiml.core, 19
miraiml.pipeline, 15
miraiml.util, 23

Index

B

BaseModel (*class in miraiml.core*), 19
BasePipelineClass (*class in miraiml.core*), 22
build_report() (*miraiml.Status method*), 15

C

clean_test_data() (*miraiml.Engine method*), 13
compose() (*in module miraiml.pipeline*), 15
Config (*class in miraiml*), 10

D

dump() (*in module miraiml.util*), 23
dump_base_model() (*in module miraiml.core*), 20

E

Engine (*class in miraiml*), 12
ensemble() (*miraiml.core.Ensembler method*), 22
Ensembler (*class in miraiml.core*), 21

F

fit() (*miraiml.core.BasePipelineClass method*), 23

G

gen_weights() (*miraiml.core.Ensembler method*), 22
get_params() (*miraiml.core.BasePipelineClass method*), 23

I

interrupt() (*miraiml.core.Ensembler method*), 22
interrupt() (*miraiml.Engine method*), 13
is_ready() (*miraiml.core.MiraiSeeker method*), 21
is_running() (*miraiml.Engine method*), 12
is_valid_filename() (*in module miraiml.util*), 24
is_valid_pipeline_name() (*in module miraiml.util*), 24

L

linear_regression_search() (*miraiml.core.MiraiSeeker method*), 21
LinearRegressionBaseliner (*class in miraiml.pipeline*), 17
load() (*in module miraiml.util*), 23
load_base_model() (*in module miraiml.core*), 20
load_test_data() (*miraiml.Engine method*), 13
load_train_data() (*miraiml.Engine method*), 13

M

miraiml (*module*), 9
miraiml.core (*module*), 19
miraiml.pipeline (*module*), 15
miraiml.util (*module*), 23
MiraiSeeker (*class in miraiml.core*), 20

N

naive_search() (*miraiml.core.MiraiSeeker method*), 21
NaiveBayesBaseliner (*class in miraiml.pipeline*), 17

O

optimize() (*miraiml.core.Ensembler method*), 22

P

parameters_features_to_dataframe() (*miraiml.core.MiraiSeeker method*), 20
predict() (*miraiml.core.BaseModel method*), 19
predict() (*miraiml.core.BasePipelineClass method*), 23
predict_proba() (*miraiml.core.BasePipelineClass method*), 23

R

random_search() (*miraiml.core.MiraiSeeker method*), 21
reconfigure() (*miraiml.Engine method*), 14
register_base_model() (*miraiml.core.MiraiSeeker method*), 20
request_status() (*miraiml.Engine method*), 14

`reset()` (*miraiml.core.MiraiSeeker method*), 20
`restart()` (*miraiml.Engine method*), 14

S

`sample_random_len()` (*in module miraiml.util*), 24
`SearchSpace` (*class in miraiml*), 9
`seek()` (*miraiml.core.MiraiSeeker method*), 21
`set_params()` (*miraiml.core.BasePipelineClass method*), 23
`shuffle_train_data()` (*miraiml.Engine method*), 14
`Status` (*class in miraiml*), 14

U

`update()` (*miraiml.core.Ensembler method*), 22