ML Tooling Documentation

Release 0.12.0

Anders Bogsnes

Mar 01, 2022

Contents

ML Tooling

Tools made clever

Welcome to ML Tooling's documentation!

Get started by installing and head over to *Quickstart*!

There are three main parts to ML Tooling:

- *Model* wraps all the functionality,
- *Transformers* has a number of pandas-compatible scikit-learn Transformers for use in scikit-learn Pipelines
- *Plotting* has a number of utility plotting functions that can be used standalone

ML Tooling is a convenient wrapper around Scikit-learn and Pandas - check the documentation there.

CHAPTER 1

Why?

ML Tooling standardises interfaces for training, saving and loading models while also providing a number of utility functions for everyday tasks such as plotting, comparing results and gridsearching.

ML Tooling was developed at Alm Brand

1.1 Table of Contents

1.1.1 Installation

Requirements **Python >= 3.7**

To install the latest released version of ml_tooling:

Using pip

Using conda

From Source

You can install ml_tooling from source just as you would install any other Python package:

```
$ pip install git+https://github.com/anders_bogsnes/ml_tooling.git
```

If you need to install the very newest development version on GitHub:

```
$ pip install -U git+https://github.com/anders_bogsnes/ml_tooling.git
```

1.1.2 Quickstart

ML Tooling **requires you to create a data class** inheriting from one of the ML Tooling dataclasses Dataset, FileDataset, SQLDataset.

You have to define two methods in your class:

- *load_training_data()*

Defines how to your training data is loaded - whether it's reading from an excel file or loading from a database. This method should read in your data and return a DataFrame containing your features and a target - usually as a numpy array or a pandas Series. This method is called the first time ML Tooling needs to gather data and is only called once.

- *load_prediction_data()*

Defines how to load your prediction data. When predicting, you have to tell ML Tooling what data to load in. Usually this takes an argument to select features for a given customer or item.

```
>>> from ml_tooling import Model
>>> from ml_tooling.data import Dataset
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.datasets import load_boston
>>> import pandas as pd
>>>
>>> class BostonData(Dataset):
...     def load_training_data(self):
...         data = load_boston()
...         return pd.DataFrame(data.data, columns=data.feature_names), data.target
...
...     # Define where to get prediction time data - returning a DataFrame
...     def load_prediction_data(self, idx):
...         data = load_boston()
...         x = pd.DataFrame(data.data, labels=data.feature_names)
...         return x.loc[idx] # Return given observation
>>>
>>> # Use your data with a given model
>>> data = BostonData()
```

To create a model, use the Model class by giving it an estimator to instantiate the class. The estimator must use scikit-learn's standard API.

```
>>> regression = Model(LinearRegression())
>>> regression
<Model: LinearRegression>
```

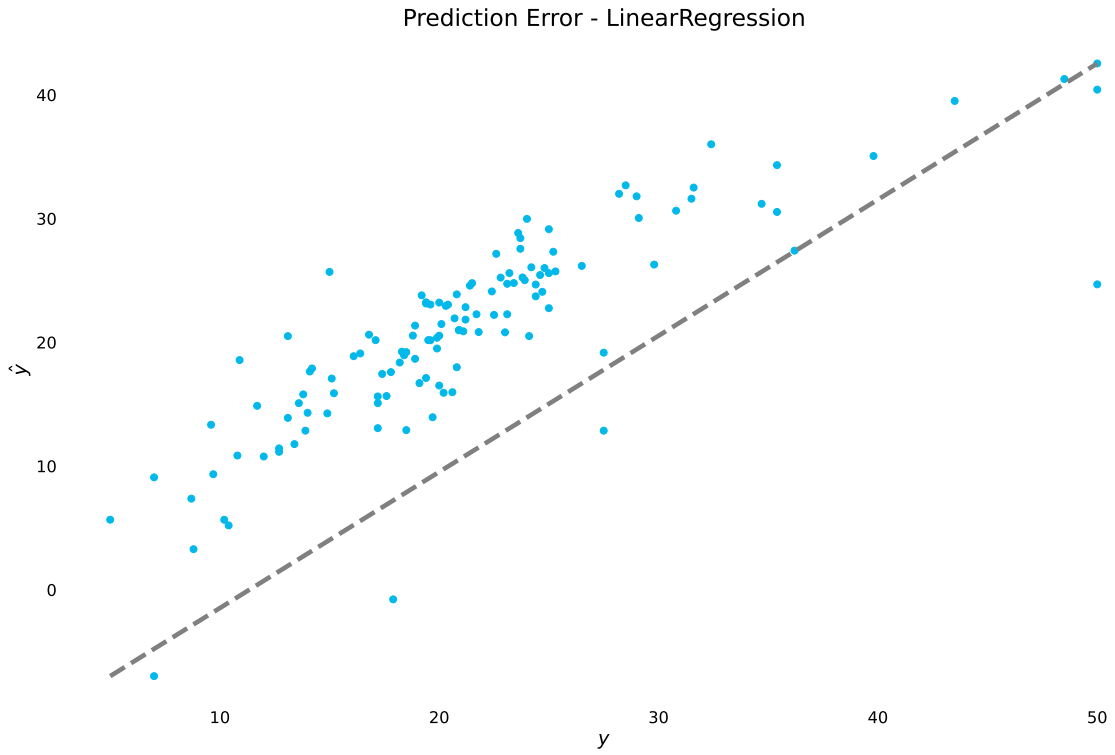
Now we can train our model. We start by splitting the data into training and test data by calling *create_train_test()*

```
>>> data.create_train_test()
<BostonData - Dataset>
>>> result = regression.score_estimator(data)
>>> result
<Result LinearRegression: {'r2': 0.68}>
```

We can plot the prediction errors:

```
>>> result.plot.prediction_error()
```

We can save and load our model:




```
>>> from ml_tooling.storage import FileStorage
>>> storage = FileStorage('./estimator_dir')
>>> file_path = regression.save_estimator(storage)
>>> my_new_model = regression.load_estimator(file_path.name, storage=storage)
>>> my_new_model
<Model: LinearRegression>
```

We can try out many different models:

```
>>> from sklearn.linear_model import Ridge, LassoLars
>>> models_to_try = [LinearRegression(), Ridge(), LassoLars()]
>>> best_model, all_results = Model.test_estimators(data,
...                                                 models_to_try,
...                                                 metrics='neg_mean_squared_error')
>>> all_results
ResultGroup(results=[<Result LinearRegression: {'neg_mean_squared_error': -22.1}>,
 ↪<Result Ridge: {'neg_mean_squared_error': -22.48}>, <Result LassoLars: {'neg_mean_
 ↪squared_error': -72.26}>])
```

We get the results in sorted order for each model and see that LinearRegression gives us the best result!

Continue to *Tutorial*

1.1.3 Tutorial

We will be using the Iris dataset for this tutorial. First we create the dataset we will be working with, by declaring a class inheriting from Dataset which will define how we load our training and prediction data. Scikit-learn comes with

handy Dataset loading utilities, so we will be using their `load_iris()` data loader function

```
>>> from sklearn.datasets import load_iris
>>> from ml_tooling.data import Dataset
>>> import pandas as pd
>>> import numpy as np
```

```
>>> class IrisData(Dataset):
...     def load_training_data(self):
...         data = load_iris()
...         target = np.where(data.target == 1, 1, 0)
...         return pd.DataFrame(data=data.data, columns=data.feature_names), target
...
...     def load_prediction_data(self, idx):
...         X, y = self.load_training_data()
...         return X.loc[idx, :].to_frame().T
>>>
>>> data = IrisData()
>>> data.create_train_test()
<IrisData - Dataset>
```

With our data object ready to go, lets move on to the model object.

```
>>> from ml_tooling import Model
>>> from sklearn.linear_model import LogisticRegression
>>>
>>> lr_clf = Model(LogisticRegression())
>>>
>>> lr_clf.score_estimator(data, metrics='accuracy')
<Result LogisticRegression: {'accuracy': 0.74}>
```

We have a few more estimators we'd like to try out and see which one performs best. We can include a `RandomForestClassifier` and a `DummyClassifier` to have a baseline metric score.

In order to have a better idea of how the models perform, we can use cross-validation and benchmark the models against each other using different metrics. The best estimator is then picked using the best mean cross-validation score

Note: Note that the results will be sorted based on the first metric passed

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.dummy import DummyClassifier
>>> estimators = [LogisticRegression(solver='lbfgs'),
...               RandomForestClassifier(n_estimators=10, random_state=42),
...               DummyClassifier(strategy="prior", random_state=42)]
>>> best_model, results = Model.test_estimators(data, estimators, metrics=['accuracy',
↪ 'roc_auc'], cv=10)
```

We can see that the results are sorted and shows us a nice repr of each model's performance

```
>>> results
ResultGroup(results=[<Result RandomForestClassifier: {'accuracy': 0.95, 'roc_auc': 0.
↪98}>, <Result LogisticRegression: {'accuracy': 0.71, 'roc_auc': 0.79}>, <Result␣
↪DummyClassifier: {'accuracy': 0.55, 'roc_auc': 0.52}>])
```

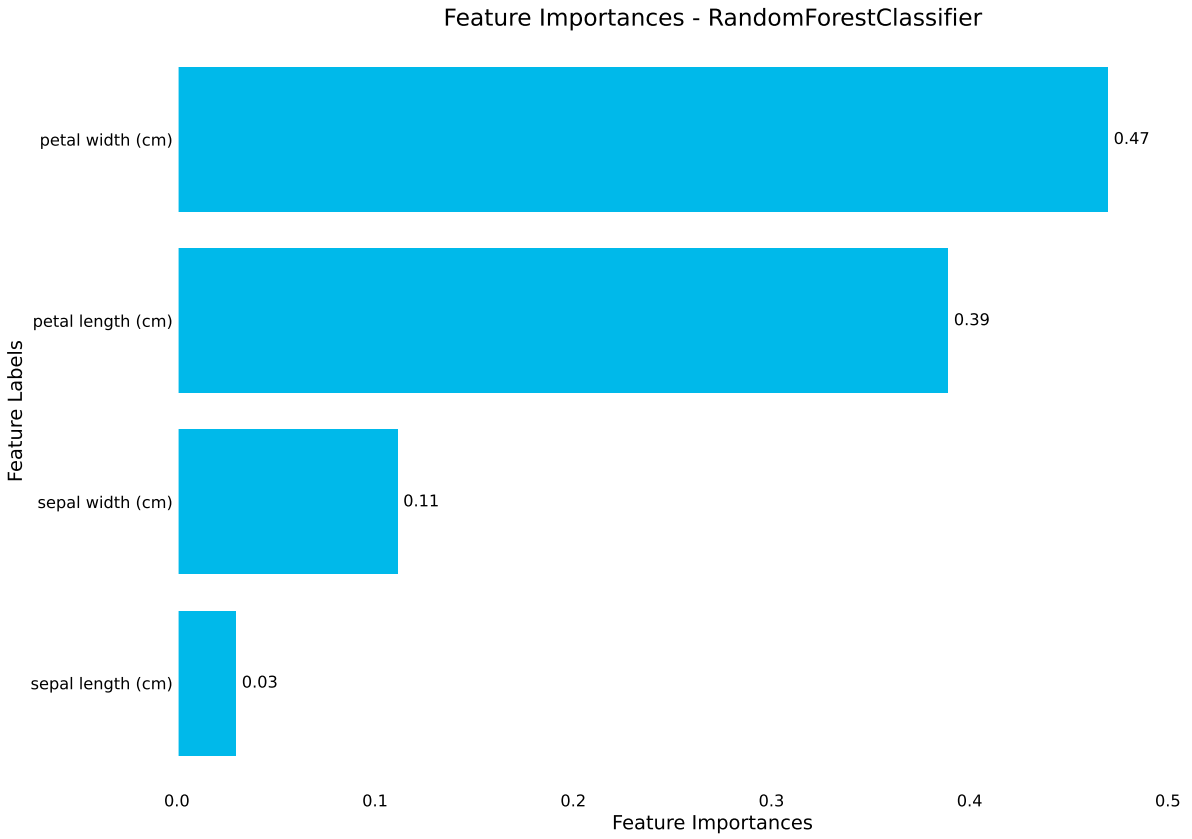
You can access the parameters via the `Result.parameters` attribute.

```
>>> results.parameters
{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_
↪depth': None, 'max_features': 'auto', 'max_leaf_nodes': None, 'max_samples': None,
↪'min_impurity_decrease': 0.0, 'min_impurity_split': None, 'min_samples_leaf': 1,
↪'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 10, 'n_jobs
↪': None, 'oob_score': False, 'random_state': 42, 'verbose': 0, 'warm_start': False}
```

From our results, the `RandomForestClassifier` looks the most promising, so we want to see if we can tune it a bit more. We can run a gridsearch over the hyperparameters using the `gridsearch()` method. We also want to log the results, so we can examine each potential model in depth, so we use the `log()` context manager, passing a log_directory where to save the files.

```
>>> # We could also use `best_model` here
>>> rf_clf = Model(RandomForestClassifier(n_estimators=10, random_state=42))
>>> with rf_clf.log('./gridsearch'):
...     best_model, results = rf_clf.gridsearch(data, {"max_depth": [3, 5, 10, 15]})
>>>
>>> results
ResultGroup(results=[<Result RandomForestClassifier: {'accuracy': 0.95}>, <Result␣
↪RandomForestClassifier: {'accuracy': 0.95}>, <Result RandomForestClassifier: {
↪'accuracy': 0.95}>, <Result RandomForestClassifier: {'accuracy': 0.93}>])
```

As the results are ordered by highest mean accuracy, we can select the first result and plot some diagnostic plots using the *.plot* accessor.



We finish up by saving our best model to a local file, so we can reload that model later

```
>>> from ml_tooling.storage import FileStorage
>>>
>>> storage = FileStorage('./estimators')
>>> saved_path = best_model.save_estimator(storage)
```

If you are interested in more examples of how to use ml-tooling, please see the project notebooks.

Continue to *Dataset*

1.1.4 Dataset

SQLDataset

Creating a SQLDataset from a table in a db

SQLDataset should be used for creating datasets based on SQL database source. SQLDatasets must be provided with a `sqlalchemy.engine.Connectable` or a valid connection string.

When writing the *load_training_data()* and *load_prediction_data()*, they must accept a connection in their arguments - this will be provided at runtime by the SQLDataset.

FileDataset

Creating a FileDataset from a csv file

When we create a FileDataset, we need to specify the location of our datafiles - this will be available in the `self.file_path` attribute. ML Tooling can A more elaborate example of using this dataset can be found at `../notebooks/Titanic Demo.ipynb`.

When a Dataset is correctly defined, you can use all the methods defined in *Dataset*

Copying Datasets

If you have two datasets defined, you can copy data from one into the other. For example, if you have defined a SQLDataset and want to copy it into a file:

This will read the data from the SQL database and write it to a csv file named `titanic.csv`

A common usecase for this is to move data from a central datastore into a local datastore, keeping two database tables in sync.

Demo Datasets

If you want to test your model on a demo datasets from Dataset loading utilities, you can use the function *load_demo_dataset()*

```
>>> from ml_tooling.data import load_demo_dataset
>>>
>>> bostondata = load_demo_dataset("boston")
>>> # Remember to setup a train test split!
>>> bostondata.create_train_test()
<BostonData - Dataset>
```

1.1.5 Model

The *Model* baseclass contains all the neat functionality of ML Tooling.

In order to take advantage of this functionality, simply wrap a model that follows the scikit-learn API using the Model class.

See also:

Refer to *Model* for a full overview of methods

We will be using scikit-learn's built-in Boston houseprices dataset to demonstrate how to use ML Tooling. We use the method *load_demo_dataset()* to load the dataset.

We then simply wrap a LinearRegression using our *Model* class and we are ready to begin!

```
>>> from ml_tooling.data import load_demo_dataset
>>>
>>> bostondata = load_demo_dataset("boston")
>>> # Remember to setup a train test split!
>>> bostondata.create_train_test()
<BostonData - Dataset>
```

Creating your model

The first thing to do after creating a dataset object is to create a model object. This is done by supplying an estimator to the *Model*.

```
>>> from ml_tooling import Model
>>> from sklearn.linear_model import LinearRegression
>>>
>>> linear = Model(LinearRegression())
>>> linear
<Model: LinearRegression>
```

Scoring your model

In order to evaluate the performance of the model use the *score_estimator()* method. This will train the estimator on the training split of our bostondata Dataset and evaluate it on the test split. If no training split has been created from the data the method will create one using the default configuration values. It returns an instance of *Result* which we can then introspect further.

```
>>> result = linear.score_estimator(bostondata)
>>> result
<Result LinearRegression: {'r2': 0.68}>
```

Testing multiple estimators

To test which estimator performs best, use the *test_estimators()* method. This method trains each estimator on the train split and evaluates the performance on the test split. It returns a new *Model* instance with the best-performing estimator with the best estimator and a *ResultGroup*.

```
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.ensemble import RandomForestRegressor
>>> best_model, results = Model.test_estimators(
...     bostondata,
...     [LinearRegression(), RandomForestRegressor(n_estimators=10, random_
↪state=1337)],
...     metrics='r2')
>>> results
ResultGroup(results=[<Result RandomForestRegressor: {'r2': 0.82}>, <Result␣
↪LinearRegression: {'r2': 0.68}>])
```

Training your model

When the best model has been found use *train_estimator()* to train the model on the full dataset set.

Note: This should be the last step before saving the model for production.

```
>>> linear.train_estimator(bostondata)
<Model: LinearRegression>
```

Predicting with your model

To make a prediction use the method *make_prediction()*. This will call the *load_prediction_data()* defined in your dataset.

```
>>> customer_id = 42
>>> linear.make_prediction(bostondata, customer_id)
   Prediction
0   25.203866
```

make_prediction() also has a parameter `proba` which will return the underlying probabilities if working on a classification problem

Defining a Feature Pipeline

It is very common to define a feature preprocessing pipeline to preprocess your data before passing it to the estimator. Using a `Pipeline` ensures that the preprocessing is "learned" on the training split and only applied on the validation split. When passing a feature_pipeline in the, *Model* will automatically create a `Pipeline` with two steps: *features* and *estimator*.

```
>>> from ml_tooling import Model
>>> from ml_tooling.transformers import DFStandardScaler
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LinearRegression
>>>
>>> feature_pipeline = Pipeline([("scaler", DFStandardScaler())])
>>> model = Model(LinearRegression(), feature_pipeline=feature_pipeline)
>>> model.estimator
Pipeline(steps=[('features', Pipeline(steps=[('scaler', DFStandardScaler())])),
                ('estimator', LinearRegression())])
```

Performing a gridsearch

To find the best hyperparameters for an estimator you can use *gridsearch()*, passing a dictionary of hyperparameters to try.

```
>>> best_estimator, results = linear.gridsearch(bostondata, { "normalize": [False,␣
↪True] })
>>> results
ResultGroup(results=[<Result LinearRegression: {'r2': 0.72}>, <Result␣
↪LinearRegression: {'r2': 0.72}>])
```

The input hyperparameters have a similar format to `GridSearchCV`, so if we are gridsearching using a `Pipeline`, we can pass hyperparameters using the same syntax.

```
>>> from sklearn.pipeline import Pipeline
>>> from ml_tooling.transformers import DFStandardScaler
>>> from ml_tooling import Model
>>>
>>> feature_pipe = Pipeline([('scale', DFStandardScaler())])
>>> pipe_model = Model(LinearRegression(), feature_pipeline=feature_pipe)
>>> best_estimator, results = pipe_model.gridsearch(bostondata, { "estimator__
↪normalize": [False, True]})
>>> results
ResultGroup(results=[<Result LinearRegression: {'r2': 0.72}>, <Result␣
↪LinearRegression: {'r2': 0.72}>])
```

Using the logging capability of Model `log()` method, we can save each result to a yaml file.

```
>>> with linear.log("./bostondata_linear"):
...     best_estimator, results = linear.gridsearch(bostondata, { "normalize": [False,
↪ True] })
```

This will generate a yaml file for each

```
created_time: 2019-10-31 17:32:08.233522
estimator:
- classname: LinearRegression
module: sklearn.linear_model.base
params:
    copy_X: true
    fit_intercept: true
    n_jobs: null
    normalize: true
estimator_path: null
git_hash: afa6def92a1e8a0ac571bec254129818bb337c49
metrics:
    r2: 0.7160133196648374
model_name: BostonData_LinearRegression
versions:
    ml_tooling: 0.9.1
    pandas: 0.25.2
    sklearn: 0.21.3
```

Performing a randomized search

Similar to the interface of the above mentioned gridsearch, you can make a more efficient but less rigorous search of the parameter space with a randomized search.

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> from ml_tooling.search import Real
>>> rand_forest = Model(RandomForestRegressor())
>>>
>>> search_space = {
...     "max_depth": [1, 3],
...     "min_weight_fraction_leaf": Real(0, 0.5),
... }
>>> best_estimator, results = rand_forest.randomsearch(bostondata, search_space, n_
↪iter=2)
>>> results
ResultGroup(results=[<Result RandomForestRegressor: {'r2': 0.83}>, <Result␣
↪RandomForestRegressor: {'r2': 0.56}>])
```

Here we specify the number of iterations *n_iter=2* just for demonstration purposes, n_iter is the number of times we sample from the parameter space to try. ML-Tooling uses skopt's Spaces to define a sampling space. You can import them from `ml_tooling.search` or from skopt directly.

When a list is given in the search space, a linear distribution is used by default, but you may also pass other distributions. ML-Tooling supports `Real`, `Integer` and `Categorical`. Each of these also support prior distributions, if more granular distributions are required.

Performing a Bayesian Search

ML-Tooling also supports Bayesian search - a stepwise search, where we build a surrogate model to estimate the effect of changing a given hyperparameter on the error. This surrogate model allows us to take steps in directions where the model thinks it can improve the error. Bayesian search is implemented using skopt and is a drop-in replacement for *`randomsearch()`*.

```
>>> from sklearn.ensemble import RandomForestRegressor
    >>> from ml_tooling.search import Real
    >>> rand_forest = Model(RandomForestRegressor())
    >>>
    >>> search_space = {
    ...     "max_depth": [1, 3],
    ...     "min_weight_fraction_leaf": Real(0, 0.5),
    ... }
    >>> best_estimator, results = rand_forest.bayesiansearch(bostondata, search_space,
↪ n_iter=2)
    >>> results
    ResultGroup(results=[<Result RandomForestRegressor: {'r2': 0.83}>, <Result␣
↪RandomForestRegressor: {'r2': 0.56}>])
>>> from ml_tooling.search import Real
>>> rand_forest = Model(RandomForestRegressor())
>>>
>>> search_space = {
...     "max_depth": [1, 3],
...     "min_weight_fraction_leaf": Real(0, 0.5),
... }
>>> best_estimator, results = rand_forest.bayesiansearch(bostondata, search_space, n_
↪iter=2)
>>> results
ResultGroup(results=[<Result RandomForestRegressor: {'r2': 0.83}>, <Result␣
↪RandomForestRegressor: {'r2': 0.56}>])
```

Storage

In order to store our estimators for later use or comparison, we use a *Storage* class and pass it to *save_estimator()*.

```
>>> from ml_tooling.storage import FileStorage
>>>
>>> estimator_dir = './estimator_dir'
>>> storage = FileStorage(estimator_dir)
>>> estimator_path = linear.save_estimator(storage)
>>> estimator_path.name
'LinearRegression_2019-10-23_13:23:22.058684.pkl'
```

The model creates a filename for the model estimator based on the current date and time and the estimator name.

We can also load the model from a storage by specifying the filename to load in the Storage directory.

```
>>> loaded_linear = linear.load_estimator(estimator_path.name, storage=storage)
>>> loaded_linear
<Model: LinearRegression>
```

Saving an estimator ready for production

You have a trained estimator ready to be saved for use in production on your filesystem.

Now users of your model package can always find your estimator through *load_production_estimator()* using the module name.

By default, if no storage is specified, ML-Tooling will save models in your current working directory in a folder called *estimators*

Configuration

To change the default configuration values, modify the `config` attributes directly:

```
>>> linear.config.RANDOM_STATE = 2
```

See also:

Refer to *Config* for a list of available configuration options

Logging

We also have the ability to log our experiments using the *Model.log()* context manager. The results will be saved in

```
>>> with linear.log('test_dir'):
...     linear.score_estimator(bostondata)
<Result LinearRegression: {'r2': 0.68}>
```

This will write a yaml file specifying attributes of the model, results, git-hash of the model and other pertinent information.

See also:

Check out *Model.log()* for more info on what is logged

Continue to *Storage*

1.1.6 Storage

ML Tooling provides different backends for storing trained models. Currently, we support local file storage, as well as Artifactory based storage.

ArtifactoryStorage

If you want to use Artifactory as a backend, first install the optional dependencies by running `pip install ml_tooling['artifactory']`.

Saving and loading an estimator from Artifactory

For more information see *ArtifactoryStorage*.

FileStorage

Saving and loading an estimator from the file system

See *FileStorage* for more information

Continue to *Plotting*

1.1.7 Plotting

Available Base plots

First we define a *Dataset* like we have done in Quickstart and Tutorial. When we score the estimator by calling *score_estimator()*, we get a *Result* back, which contains a number of handy plotting features.

To use the visualizations, access them using the *.plot* accessor on the *Result* object:

```
>>> result.plot.residuals()
```

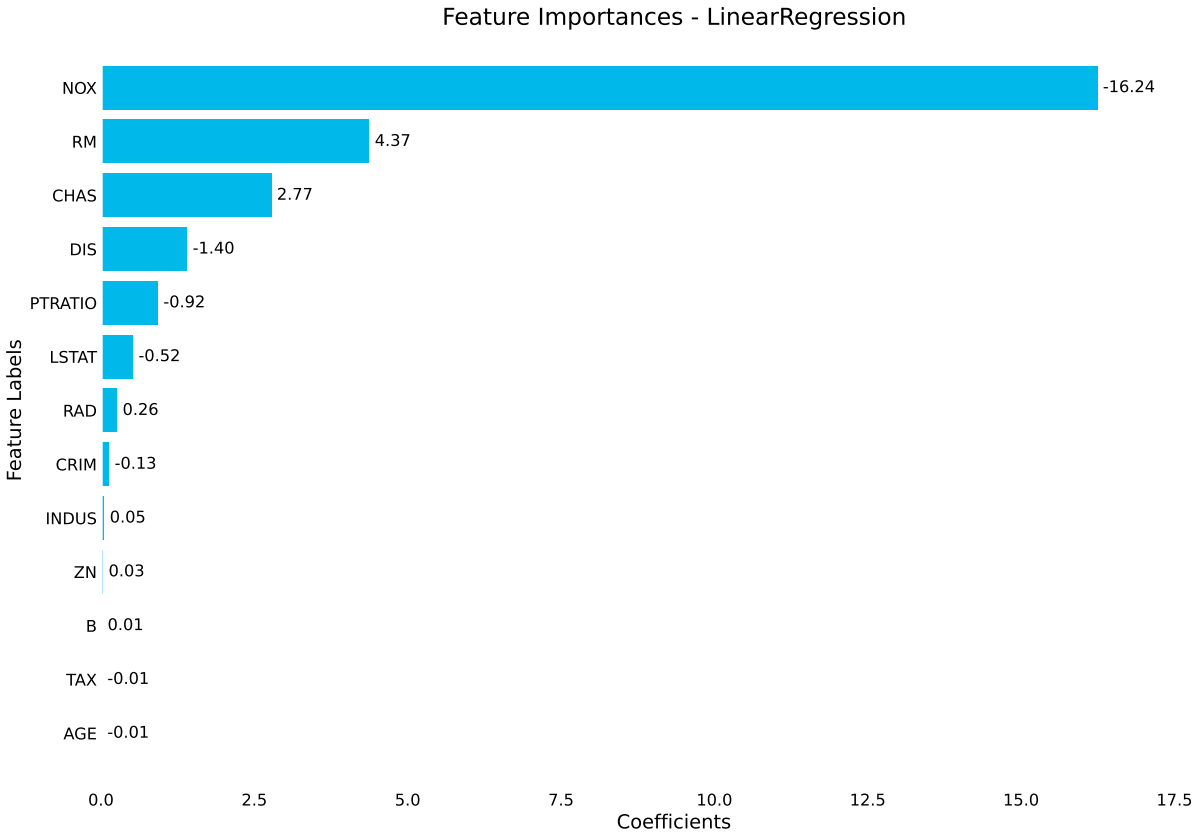
```
>>> result.plot.learning_curve()
```

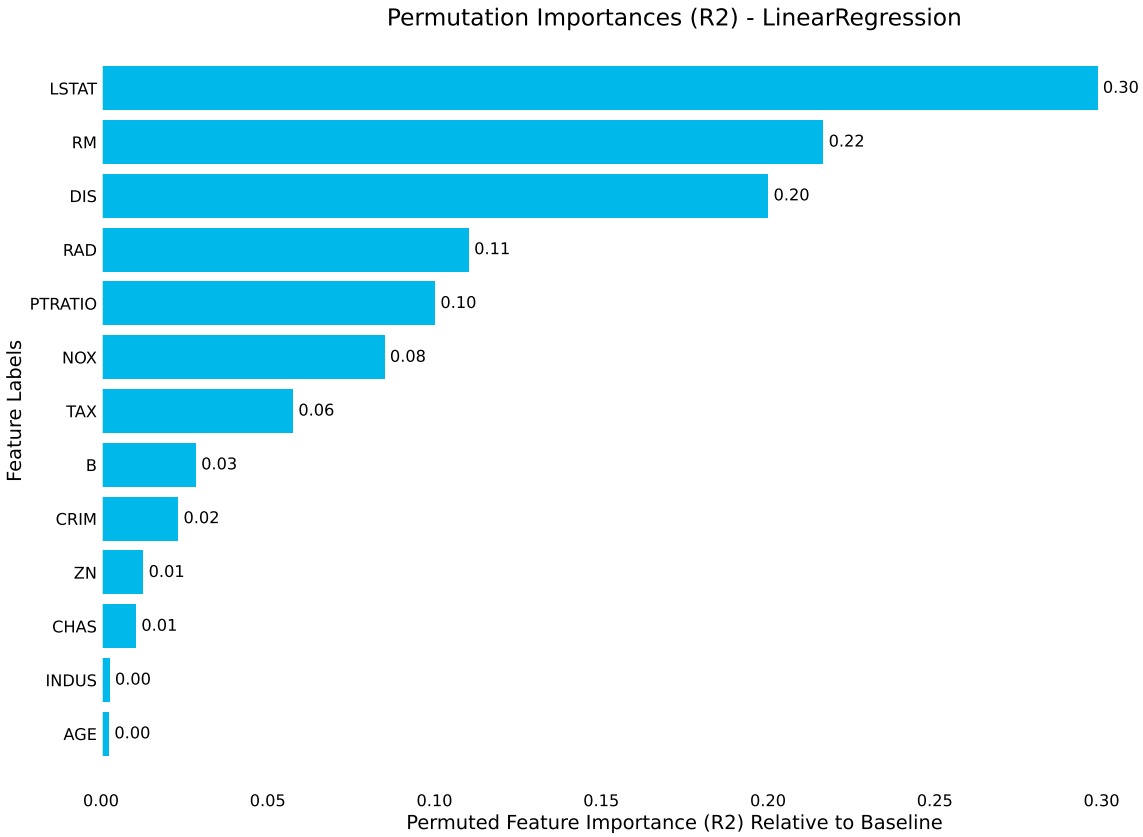
Any visualization listed here also has a functional counterpart in *ml_tooling.plots*. E.g if you want to use the function for plotting a confusion matrix without using the *Result* class

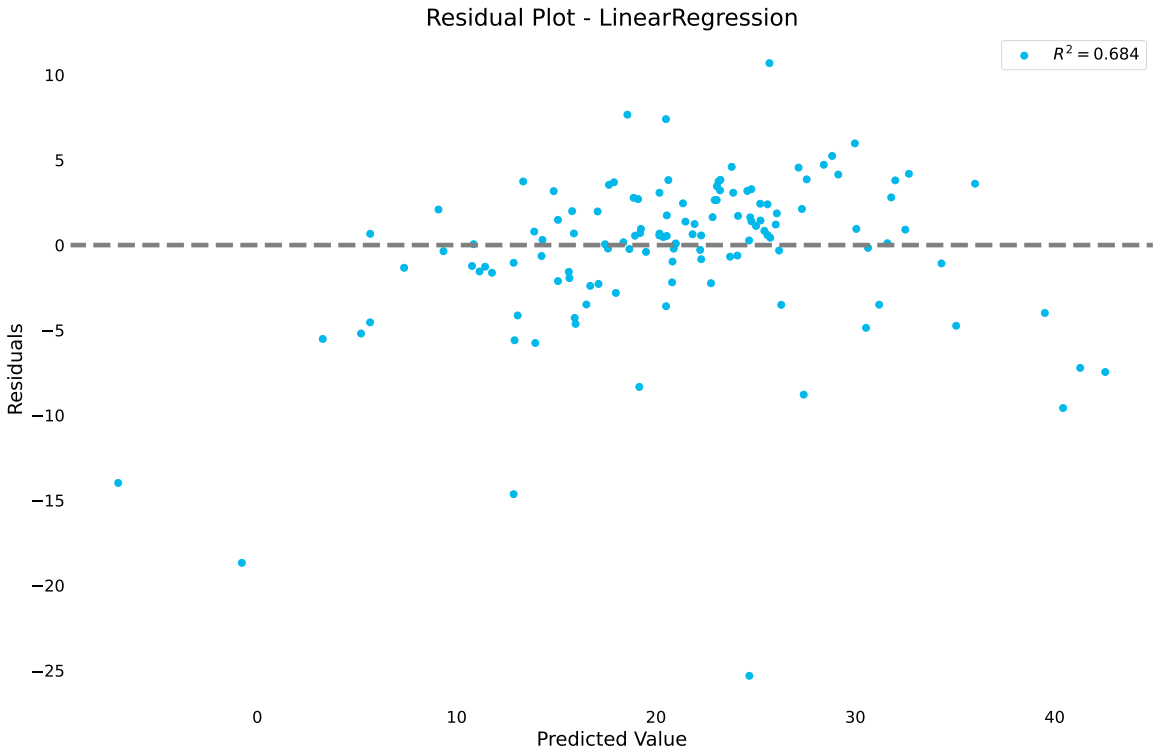
```
>>> from ml_tooling.plots import plot_confusion_matrix
```

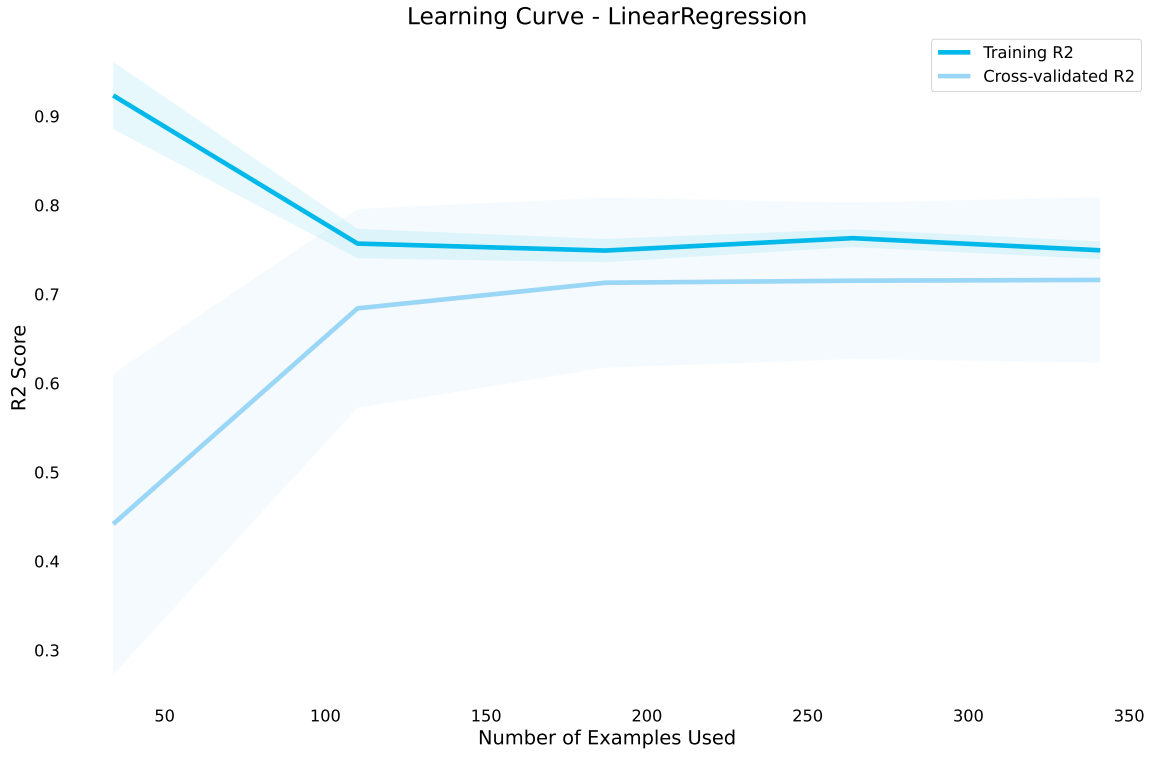
These functional counterparts all mirror the sklearn metrics api, taking y_target and y_pred as arguments:

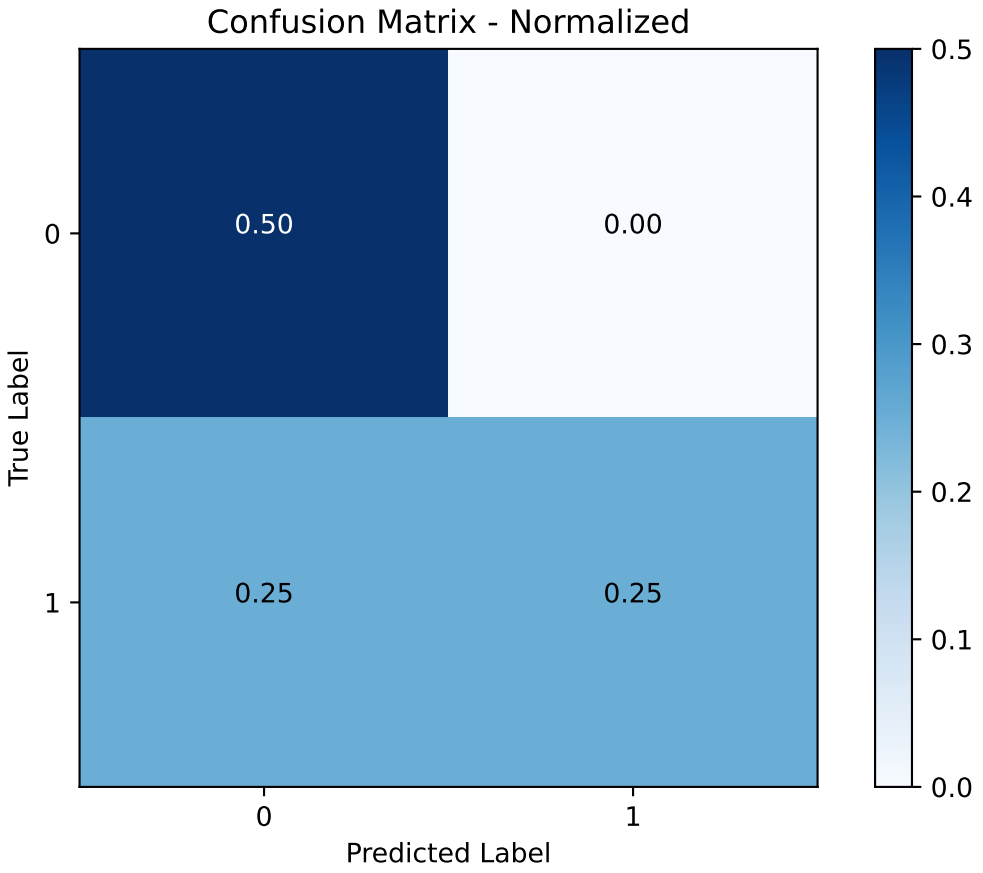
```
>>> from ml_tooling.plots import plot_confusion_matrix
>>> import numpy as np
>>>
>>> y_true = np.array([1, 0, 1, 0])
>>> y_pred = np.array([1, 0, 0, 0])
>>> plot_confusion_matrix(y_true, y_pred)
```


Feature Importances - LinearRegression
NOX
RM
CHAS
DIS
PTRATIO
LSTAT
RAD
CRIM
INDUS
ZN
B
TAX
AGE
-16.24
4.37
2.77
-1.40
-0.92
-0.52
0.26
-0.13
0.05
0.03
0.01
-0.01
-0.01
Feature Labels
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Coefficients


Permutation Importances (R2) - LinearRegression
LSTAT
RM
DIS
RAD
PTRATIO
NOX
TAX
B
CRIM
ZN
CHAS
INDUS
AGE
0.30
0.22
0.20
0.11
0.10
0.08
0.06
0.03
0.02
0.01
0.01
0.00
0.00
Feature Labels
0.00
0.05
0.10
0.15
0.20
0.25
0.30
Permuted Feature Importance (R2) Relative to Baseline


Residual Plot - LinearRegression
$R^2 = 0.684$
Residuals
Predicted Value
10
5
0
−5
−10
−15
−20
−25
0
10
20
30
40


Learning Curve - LinearRegression
Training R2
Cross-validated R2
R2 Score
Number of Examples Used
0.9
0.8
0.7
0.6
0.5
0.4
0.3
50
100
150
200
250
300
350


Confusion Matrix - Normalized
0.50
0.00
0.25
0.25
0
1
True Label
0
1
Predicted Label
0.5
0.4
0.3
0.2
0.1
0.0

Available Base plots

- ***feature_importance()*** Uses the estimator's learned coefficients or learned feature importance in the case of RandomForest to plot the relative importance of each feature. Note that for most usecases, permutation importance is going to be more accurate, but is also more computationally expensive. Pass a *class_index* parameter to select which class to plot for in a multi-class setting
- ***permutation_importance()*** Uses random permutation to calculate feature importance by randomly permuting each column and measuring the difference in the model metric against the baseline.
- ***learning_curve()*** Draws a learning curve, showing how number of training examples affects model performance. Can also be used to diagnose overfitting and underfitting by examining training and validation set performance
- ***validation_curve()*** Visualizes the impact of a given hyperparameter on the model metric by plotting a range of different hyperparameter values

Available Classifier plots

- ***roc_curve()*** Visualize a ROC curve for a classification model. Shows the relationship between the True Positive Rate and the False Positive Rate. Supports multi-class classification problems
- ***confusion_matrix()*:** Visualize a confusion matrix for a classification model. Shows the distribution of predicted labels vs actual labels Supports multi-class classification problems
- ***lift_curve()*** Visualizes how much of the target class we capture by setting different thresholds for probability. Supports multi-class classification problems
- ***precision_recall_curve()*** Visualize a Precision-Recall curve for a classification estimator. Estimator must implement a *predict_proba* method. Supports multi-class classification problems

Available Regression Plots

- ***prediction_error()*:** Visualizes prediction error of a regression model. Shows how far away each prediction is from the correct prediction for that point
- ***residuals()*:** Visualizes residuals of a regression model. Shows the distribution of noise that couldn't be fitted.

1.1.8 Data Plotting

`Dataset` also define plotting methods under the *.plot* accessor.

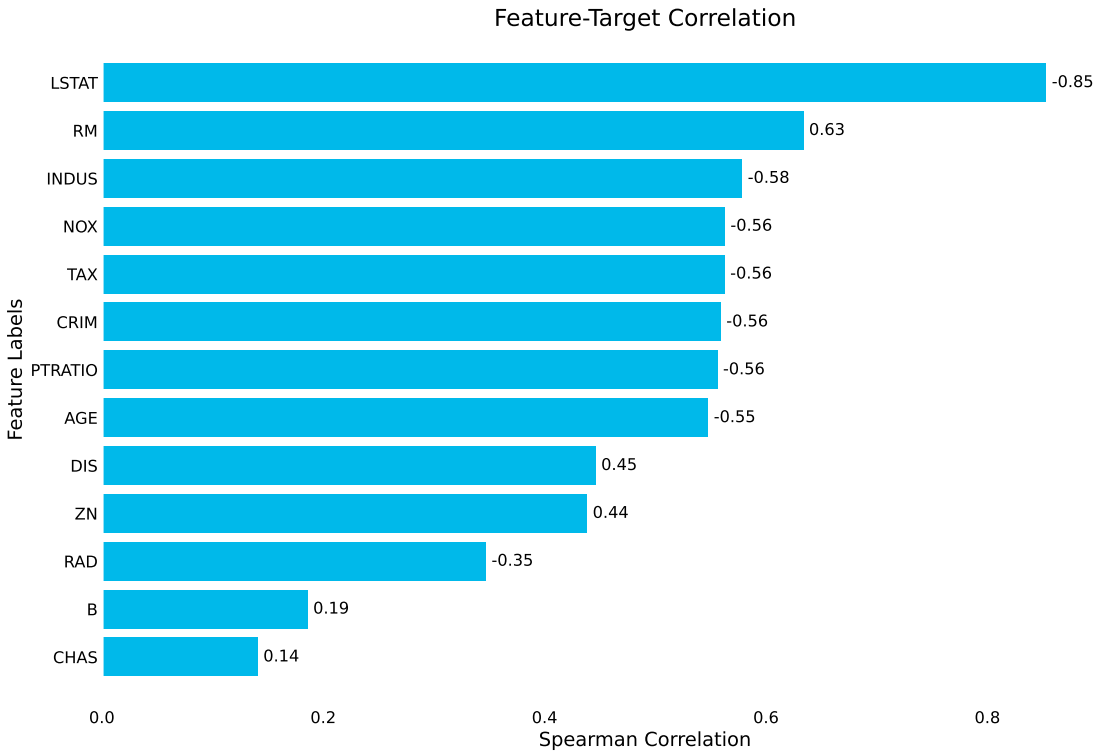
These plots are intended to help perform exploratory data analysis to inform the choices of preprocessing and models

These plot methods are used the same way as the result plots

Optionally, you can pass a preprocessing `Pipeline` to the plotter to preprocess the data before plotting. This can be useful if you want to check that the preprocessing is handling all the NaNs, or if you want to visualize computed columns.

Available Data Plots

- ***target_correlation()*:** Visualizes the correlations between each feature and the target variable. The size of the correlation can indicate important features, but can also hint at data leakage if the correlation is too strong.



- ***missing_data()*:** Visualizes percentage of missing data for each column in the dataset. If no columns have missing data, will simply show an empty plot.

Continue to *Transformers*

1.1.9 Transformers

One great feature of scikit-learn is the concept of the `Pipeline` alongside transformers

By default, scikit-learn's transformers will convert a pandas `DataFrame` to numpy arrays - losing valuable column information in the process. We have implemented a number of transformers that accept a pandas `DataFrame` and return a pandas `DataFrame`.

Select

A column selector - Provide a list of columns to be passed on in the pipeline

Example

Pass a list of column names to be selected

```
>>> from ml_tooling.transformers import Select
>>> import pandas as pd
>>> df = pd.DataFrame({
```

(continues on next page)

(continued from previous page)

```
...     "id": [1, 2, 3, 4],
...     "status": ["OK", "Error", "OK", "Error"],
...     "sales": [2000, 3000, 4000, 5000]
... })
>>> select = Select(['id', 'status'])
>>> select.fit_transform(df)
   id status
0   1     OK
1   2  Error
2   3     OK
3   4  Error
```

FillNA

Fills NA values with given value or strategy. Either a value or a strategy has to be supplied.

Examples

You can pass any value to replace NaNs with

```
>>> from ml_tooling.transformers import FillNA
>>> import numpy as np
>>> df = pd.DataFrame({
...         "id": [1, 2, 3, 4],
...         "sales": [2000, 3000, 4000, np.nan]
... })
>>> fill_na = FillNA(value = 0)
>>> fill_na.fit_transform(df)
   id   sales
0   1  2000.0
1   2  3000.0
2   3  4000.0
3   4     0.0
```

You can also use one of the built-in strategies.

- `mean`
- `median`
- `most_freq`
- `max`
- `min`

```
>>> fill_na = FillNA(strategy='mean')
>>> fill_na.fit_transform(df)
   id   sales
0   1  2000.0
1   2  3000.0
2   3  4000.0
3   4  3000.0
```

In addition, FillNa will indicate if a value in a column was missing if you set *indicate_nan=True*. This creates a new column of 1 and 0 indicating missing values

```
>>> fill_na = FillNA(strategy='mean', indicate_nan=True)
>>> fill_na.fit_transform(df)
   id   sales  id_is_nan  sales_is_nan
0   1  2000.0          0             0
1   2  3000.0          0             0
2   3  4000.0          0             0
3   4  3000.0          0             1
```

ToCategorical

Performs one-hot encoding of categorical values through `pandas.Categorical`. All categorical values not found in training data will be set to 0

Example

```
>>> from ml_tooling.transformers import ToCategorical
>>> df = pd.DataFrame({
...     "status": ["OK", "Error", "OK", "Error"]
... })
>>> onehot = ToCategorical()
>>> onehot.fit_transform(df)
   status_Error  status_OK
0             0          1
1             1          0
2             0          1
3             1          0
```

FuncTransformer

Applies a given function to each column

Example

We can use any arbitrary function that accepts a `pandas.Series` - under the hood, FuncTransformer uses `apply()`

```
>>> from ml_tooling.transformers import FuncTransformer
>>> df = pd.DataFrame({
...     "status": ["OK", "Error", "OK", "Error"]
... })
>>> uppercase = FuncTransformer(lambda x: x.str.upper())
>>> uppercase.fit_transform(df)
  status
0     OK
1  ERROR
2     OK
3  ERROR
```

FuncTransformer also supports passing keyword arguments to the function

```
>>> from ml_tooling.transformers import FuncTransformer
>>> def custom_func(input, word1, word2):
...     result = ""
...     if input == "OK":
...         result = word1
...     elif input == "Error":
...         result = word2
...     return result
>>> def wrapper(df, word1, word2):
...   return df.apply(custom_func,args=(word1,word2))
>>> df = pd.DataFrame({
...      "status": ["OK", "Error", "OK", "Error"]
... })
>>> kwargs = {'word1': 'Okay','word2': 'Fail'}
>>> wordchange = FuncTransformer(wrapper,**kwargs)
>>> wordchange.fit_transform(df)
  status
0   Okay
1   Fail
2   Okay
3   Fail
```

Binner

Bins numerical data into supplied bins. Bins are passed on to `pandas.cut()`

Example

Here we want to bin our sales data into 3 buckets

```
>>> from ml_tooling.transformers import Binner
>>> df = pd.DataFrame({
...     "sales": [1500, 2000, 2250, 7830]
... })
>>> binned = Binner(bins=[0, 1000, 2000, 8000])
>>> binned.fit_transform(df)
          sales
0  (1000, 2000]
1  (1000, 2000]
2  (2000, 8000]
3  (2000, 8000]
```

Renamer

Renames columns to be equal to the passed list - must be in order

Example

```
>>> from ml_tooling.transformers import Renamer
>>> df = pd.DataFrame({
...      "Total Sales": [1500, 2000, 2250, 7830]
```

(continues on next page)

(continued from previous page)

```
... })
>>> rename = Renamer(['sales'])
>>> rename.fit_transform(df)
   sales
0   1500
1   2000
2   2250
3   7830
```

DateEncoder

Adds year, month, day and week columns based on a datefield. Each date type can be toggled in the initializer

Example

```
>>> from ml_tooling.transformers import DateEncoder
>>> df = pd.DataFrame({
...     "sales_date": [pd.to_datetime('2018-01-01'), pd.to_datetime('2018-02-02')]
... })
>>> dates = DateEncoder(week=False)
>>> dates.fit_transform(df)
   sales_date_day  sales_date_month  sales_date_year
0               1                 1             2018
1               2                 2             2018
```

FreqFeature

Converts a column into a normalized frequency

Example

```
>>> from ml_tooling.transformers import FreqFeature
>>> df = pd.DataFrame({
...     "sales_category": ['Sale', 'Sale', 'Not Sale']
... })
>>> freq = FreqFeature()
>>> freq.fit_transform(df)
   sales_category
0        0.666667
1        0.666667
2        0.333333
```

DFFeatureUnion

A FeatureUnion equivalent for DataFrames. Concatenates the result of multiple transformers

Example

```
>>> from ml_tooling.transformers import FreqFeature, Binner, Select, DFFeatureUnion
>>> from sklearn.pipeline import Pipeline
>>> df = pd.DataFrame({
...     "sales_category": ['Sale', 'Sale', 'Not Sale', 'Not Sale'],
...     "sales": [1500, 2000, 2250, 7830]
... })
>>> freq = Pipeline([
...     ('select', Select('sales_category')),
...     ('freq', FreqFeature())
... ])
>>> binned = Pipeline([
...     ('select', Select('sales')),
...     ('bin', Binner(bins=[0, 1000, 2000, 8000]))
...     ])
>>> union = DFFeatureUnion([
...    ('sales_category', freq),
...    ('sales', binned)
... ])
>>> union.fit_transform(df)
   sales_category         sales
0             0.5  (1000, 2000]
1             0.5  (1000, 2000]
2             0.5  (2000, 8000]
3             0.5  (2000, 8000]
```

DFRowFunc

Row-wise operation on `pandas.DataFrame`. Strategy can either be one of the predefined or a callable. If some elements in the row are NaN these elements are ignored for the built-in strategies. The built-in strategies are 'sum', 'min' and 'max'

Example

```
>>> from ml_tooling.transformers import DFRowFunc
>>> df = pd.DataFrame({
...    "number_1": [1, np.nan, 3, 4],
...    "number_2": [1, 3, 2, 4]
... })
>>> rowfunc = DFRowFunc(strategy = 'sum')
>>> rowfunc.fit_transform(df)
     0
0  2.0
1  3.0
2  5.0
3  8.0
```

You can also use any callable that takes a `pandas.Series`

```
>>> rowfunc = DFRowFunc(strategy = np.mean)
>>> rowfunc.fit_transform(df)
     0
0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0
2  2.5
3  4.0
```

Binarize

Convenience transformer which returns 1 where the column value is equal to given value else 0.

Example

```
>>> from ml_tooling.transformers import Binarize
>>> df = pd.DataFrame({
...     "number_1": [1, np.nan, 3, 4],
...     "number_2": [1, 3, 2, 4]
... })
>>> binarize = Binarize(value = 3)
>>> binarize.fit_transform(df)
   number_1  number_2
0         0         0
1         0         1
2         1         0
3         0         0
```

RareFeatureEncoder

Replaces categories with a value, if they occur less than a threshold. - Using `pandas.Series.value_counts()`. The fill value can be any value and the threshold can be either a percent or int value.

The column names needs to be identical when using Train & Test dataset

The Transformer does not count NaN.

Example

```
>>> from ml_tooling.transformers import RareFeatureEncoder
>>> df = pd.DataFrame({
...         "categorical_a": [1, "a", "a", 2, "b", np.nan],
...         "categorical_b": [1, 2, 2, 3, 3, 3],
...         "categorical_c": [1, "a", "a", 2, "b", "b"],
... })

>>> rare = RareFeatureEncoder(threshold=2, fill_rare="Rare")
>>> rare.fit_transform(df)
  categorical_a categorical_b categorical_c
0          Rare          Rare          Rare
1             a             2             a
2             a             2             a
3          Rare             3          Rare
4          Rare             3             b
5           NaN             3             b
```

1.2 API Reference

If you want to see information on a specific method or function, you can find that here

1.2.1 API

Model

A wrapper for scikit-learn based estimators Implements all the base functionality needed to create the wrapper

class ml_tooling.baseclass.**Model**(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], feature_pipeline: sklearn.pipeline.Pipeline = None*)

Wrapper class for Estimators

Parameters

- **estimator** (*Estimator*) – Any scikit-learn compatible estimator
- **feature_pipeline** (Pipeline) – Optionally pass a feature preprocessing Pipeline. Model will automatically insert the estimator into a preprocessing pipeline

bayesiansearch(*data: ml_tooling.data.base_data.Dataset, param_distributions: dict, metrics: Union[str, List[str]] = 'default', cv: Optional[int] = None, n_iter: int = 10, refit: bool = True*) → Tuple[ml_tooling.baseclass.Model, ml_tooling.result.result_group.ResultGroup]

Runs a cross-validated Bayesian Search on the estimator with a randomized sampling of the passed parameter distributions

Parameters

- **data** (Dataset) – An instance of a DataSet object
- **param_distributions** (*dict*) – Parameter distributions to use for randomizing search. Should be a dictionary of param_names -> one of - ml_tooling.search.Integer - ml_tooling.search.Categorical - ml_tooling.search.Real
- **metrics** (*str, list of str*) – Metrics to use for scoring. "default" sets metric equal to self.default_metric. First metric is used to sort results.
- **cv** (*int, optional*) – Cross validation to use. Defaults to value in config.CROSS_VALIDATION
- **n_iter** (*int*) – Number of parameter settings that are sampled.
- **refit** (*bool*) – Whether or not to refit the best model

Returns

- **best_estimator** (*Model*) – Best estimator as found by the Bayesian Search
- **result_group** (*ResultGroup*) – ResultGroup object containing each individual score

default_metric

Defines default metric based on whether or not the estimator is a regressor or classifier. Then CLASSIFIER_METRIC or CLASSIFIER_METRIC is returned.

Returns Name of the metric

Return type str

gridsearch(*data: ml_tooling.data.base_data.Dataset*, *param_grid: dict*, *metrics: Union[str, List[str]] = 'default'*, *cv: Optional[int] = None*, *refit: bool = True*) → Tuple[ml_tooling.baseclass.Model, ml_tooling.result.result_group.ResultGroup]

Runs a cross-validated gridsearch on the estimator with the passed in parameter grid.

Parameters

- **data** (Dataset) – An instance of a DataSet object
- **param_grid** (*dict*) – Parameters to use for grid search
- **metrics** (*str, list of str*) – Metrics to use for scoring. "default" sets metric equal to `self.default_metric`. First metric is used to sort results.
- **cv** (*int, optional*) – Cross validation to use. Defaults to value in `config.CROSS_VALIDATION`
- **refit** (*bool*) – Whether or not to refit the best model

Returns

- **best_estimator** (*Model*) – Best estimator as found by the gridsearch
- **result_group** (*ResultGroup*) – ResultGroup object containing each individual score

static list_estimators(*storage: ml_tooling.storage.base.Storage*) → List[pathlib.Path]

Gets a list of estimators from the given Storage

Parameters **storage** (Storage) – Storage class to list the estimators with

Example

storage = FileStorage('path/to/estimators_dir') estimator_list = Model.list_estimators(storage)

Returns list of Paths

Return type List[pathlib.Path]

classmethod load_estimator(*path: Union[str, pathlib.Path]*, *storage: ml_tooling.storage.base.Storage = None*) → ml_tooling.baseclass.Model

Instantiates the class with a joblib pickled estimator.

Parameters

- **storage** (Storage) – Storage class to load the estimator with
- **path** (*str, pathlib.Path, optional*) – Path to estimator pickle file

Example

We can load a trained estimator from disk:

```
storage = FileStorage('path/to/dir')
my_estimator = Model.load_estimator('my_model.pkl', storage=storage)
```

We now have a trained estimator loaded.

We can also use the default storage:

```
my_estimator = Model.load_estimator('my_model.pkl')
```

This will use the default FileStorage defined in Model.config.default_storage

Returns Instance of Model with a saved estimator

Return type *Model*

classmethod load_production_estimator(*module_name: str*)
Loads a model from a python package. Given that the package is an ML-Tooling package, this will load the production model from the package and create an instance of Model with that package

Parameters **module_name** (*str*) – The name of the package to load a model from

log(*run_directory: str*)
log() is a context manager that lets you turn on logging for any scoring methods that follow. You can pass a log_dir to specify a subdirectory to store the estimator in. The output is a yaml file recording estimator parameters, package version numbers, metrics and other useful information

Parameters **run_directory** (*str*) – Name of the folder to save the details in

Example

If we want to log an estimator run in the *score* folder we can write:

```
with estimator.log('score'):
    estimator.score_estimator
```

This will save the results of *estimator.score_estimator()* to *runs/score/*

make_prediction(*data: ml_tooling.data.base_data.Dataset*, **args*, *proba: bool = False*, *threshold: float = None*, *use_index: bool = False*, *use_cache: bool = False*, ***kwargs*) → pandas.core.frame.DataFrame
Makes a prediction given an input. For example a customer number. Calls *load_prediction_data(*args)* and passes resulting data to *predict()* on the estimator

Parameters

- **data** (Dataset) – an instantiated Dataset object
- **proba** (*bool*) – Whether prediction is returned as a probability or not. Note that the return value is an n-dimensional array where n = number of classes
- **threshold** (*float*) – Threshold to use for predicting a binary class
- **use_index** (*bool*) – Whether the index from the prediction data should be used for the result.
- **use_cache** (*bool*) – Whether or not to use the cached data in dataset to make predictions. Useful for seeing probability distributions of the model

Returns A DataFrame with a prediction per row.

Return type pd.DataFrame

randomsearch(*data: ml_tooling.data.base_data.Dataset*, *param_distributions: dict*, *metrics: Union[str, List[str]] = 'default'*, *cv: Optional[int] = None*, *n_iter: int = 10*, *refit: bool = True*) → Tuple[ml_tooling.baseclass.Model, ml_tooling.result.result_group.ResultGroup]
Runs a cross-validated randomsearch on the estimator with a randomized sampling of the passed parameter distributions

Parameters

- **data** (Dataset) – An instance of a DataSet object

- **param_distributions** (*dict*) – Parameter distributions to use for randomizing search
- **metrics** (*str, list of str*) – Metrics to use for scoring. "default" sets metric equal to `self.default_metric`. First metric is used to sort results.
- **cv** (*int, optional*) – Cross validation to use. Defaults to value in `config.CROSS_VALIDATION`
- **n_iter** (*int*) – Number of parameter settings that are sampled.
- **refit** (*bool*) – Whether or not to refit the best model

Returns

- **best_estimator** (*Model*) – Best estimator as found by the randomsearch
- **result_group** (*ResultGroup*) – ResultGroup object containing each individual score

save_estimator(*storage: ml_tooling.storage.base.Storage = None*, *prod=False*) → pathlib.Path

Saves the estimator as a binary file.

Parameters

- **storage** (Storage) – Storage class to save the estimator with
- **prod** (*bool*) – Whether this is a production model to be saved

Example

If we have trained an estimator and we want to save it to disk we can write:

```
storage = FileStorage('/path/to/save/dir')
model = Model(LinearRegression())
saved_filename = model.save_estimator(storage)
```

to save in the given folder.

Returns The path to where the estimator file was saved

Return type pathlib.Path

score_estimator(*data: ml_tooling.data.base_data.Dataset*, *metrics: Union[str, List[str]] = 'default'*, *cv: Optional[int] = False*) → ml_tooling.result.result.Result

Scores the estimator based on training data from *data* and validates based on validation data from *data*.

Defaults to no cross-validation. If you want to cross-validate the results, pass number of folds to cv. If cross-validation is used, *score_estimator* only cross-validates on training data and doesn't use the validation data.

If the dataset does not have a train set, it will create one using the default config.

Returns a `Result` object containing all result parameters

Parameters

- **data** (Dataset) – An instantiated Dataset object with create_train_test called
- **metrics** (*string, list of strings*) – Metric or metrics to use for scoring the estimator. Any sklearn metric string
- **cv** (*int, optional*) – Whether or not to use cross validation. Number of folds if an int is passed If False, don't use cross validation

Returns A Result object that contains the results of the scoring

Return type *Result*

classmethod test_estimators(*data: ml_tooling.data.base_data.Dataset, estimators: Sequence[Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline]], feature_pipeline: sklearn.pipeline.Pipeline = None, metrics: Union[str, List[str]] = 'default', cv: Union[int, bool] = False, log_dir: str = None, refit: bool = False*) → Tuple[ml_tooling.baseclass.Model, ml_tooling.result.result_group.ResultGroup]

Trains each estimator passed and returns a sorted list of results

Parameters

- **data** (*Dataset*) – An instantiated Dataset object with train_test data
- **estimators** (*Sequence[Estimator]*) – List of estimators to train
- **feature_pipeline** (*Pipeline*) – A pipeline for transforming features
- **metrics** (*str, list of str*) – Metric or list of metrics to use in scoring of estimators
- **cv** (*int, bool*) – Whether or not to use cross-validation. If an int is passed, use that many folds
- **log_dir** (*str, optional*) – Where to store logged estimators. If None, don't log
- **refit** (*bool*) – Whether or not to refit the best model on all the training data

Returns

Return type List of Result objects

to_dict() → List[dict]

Serializes the estimator to a dictionary

Returns

Return type List of dicts

train_estimator(*data: ml_tooling.data.base_data.Dataset*) → ml_tooling.baseclass.Model

Loads all training data and trains the estimator on all data. Typically used as the last step when estimator tuning is complete.

> **Warning:** This will set self.result attribute to None. This method trains the estimator using all the data, so there is no validation data to measure results against

Returns Returns an estimator trained on all the data, with no train-test split

Return type *Model*

Datasets

class ml_tooling.data.**FileDataset**(*path: Union[str, pathlib.Path]*)

An Abstract Base Class for use in creating Filebased Datasets. This class is intended to be subclassed and must provide a *load_training_data()* and *load_prediction_data()* method.

FileDataset takes a path as its initialization argument, pointing to a file which must be a filetype supported by Pandas, such as csv, parquet etc. The extension determines the pandas method used to read and write the data

Instantiates a Filedataset pointing at a given path.

Parameters **path** (*Pathlike*) – Path to location of file

load_prediction_data(**args*, ***kwargs*) → pandas.core.frame.DataFrame
Used to load prediction data for a given idx - returns features

load_training_data(**args*, ***kwargs*) → Tuple[pandas.core.frame.DataFrame, Union[pandas.core.series.Series, numpy.ndarray]]
Used to load the full training dataset - returns features and targets

read_file(***kwargs*)
Read the data from the passed file path

Parameters **kwargs** (*dict*) – Kwargs are passed to the relevant `pd.read_*()` method for the given extension

class ml_tooling.data.**SQLDataset**(*conn: Union[str, sqlalchemy.engine.interfaces.Connectable], schema: Optional[str], **kwargs*)
An Abstract Base Class for use in creating SQL Datasets. This class is intended to be subclassed and must provide a *load_training_data()* and *load_prediction_data()* method.

These methods must accept a *conn* argument which is an instance of a SQLAlchemy connection. This connection will be passed to the method by the SQLDataset at runtime.

table
SQLAlchemy table definition to use when loading the dataset. Is the table that will be copied when using *.copy_to* and should be the canonical definition of the feature set. Do not define a schema - that is set at runtime

Type sa.Table

Instantiates a dataset with the necessary arguments to connect to the database.

Parameters

- **conn** (*Connectable*) – Either a valid DB_URL string or an engine to connect to the database
- **schema** (*str*) – A string naming the schema to use - allows for swapping schemas at runtime
- **kwargs** (*dict*) – Kwargs are passed to *create_engine* if conn is a string

copy_to(*target: ml_tooling.data.sql.SQLDataset*) → ml_tooling.data.sql.SQLDataset
Copies data from one database table into other. This will truncate the table and load new data in.

Parameters **target** (*SQLDataset*) – A SQLDataset object representing the table you want to copy the data into

Returns The target dataset to copy to

Return type *SQLDataset*

create_connection() → sqlalchemy.engine.base.Connection
Instantiates a connection to be used in reading and writing to the database.

Ensures that connections are closed properly and dynamically inserts the schema into the database connection

Returns An open connection to the database, with a dynamically defined schema

Return type sa.engine.Connection

load_prediction_data(*idx*, *conn*, **args*, ***kwargs*) → pandas.core.frame.DataFrame
Used to load prediction data for a given idx - returns features

load_training_data(*conn*, **args*, ***kwargs*) → Tuple[pandas.core.frame.DataFrame, Union[pandas.core.series.Series, numpy.ndarray]]
Used to load the full training dataset - returns features and targets

class ml_tooling.data.**Dataset**
Baseclass for creating Datasets. Subclass Dataset and provide a *load_training_data()* and *load_prediction_data()* method

create_train_test(*stratify: bool = False*, *shuffle: bool = True*, *test_size: float = 0.25*, *seed: int = 42*) → ml_tooling.data.base_data.Dataset
Creates a training and testing dataset and storing it on the data object.

Parameters

- **stratify** (*DataType, optional*) – What to stratify the split on. Usually y if given a classification problem
- **shuffle** – Whether or not to shuffle the data
- **test_size** – What percentage of the data will be part of the test set
- **seed** – Random seed for train_test_split

Returns

Return type self

load_prediction_data(**args*, ***kwargs*) → pandas.core.frame.DataFrame
Abstract method to be implemented by the user. Defines data to be used at prediction time, defined as a DataFrame

Returns DataFrame of input features to get a prediction

Return type pd.DataFrame

load_training_data(**args*, ***kwargs*) → Tuple[pandas.core.frame.DataFrame, Union[pandas.core.series.Series, numpy.ndarray]]
Abstract method to be implemented by user. Defines data to be used at training time where X is a dataframe and y is a numpy array

Returns **x, y** – Training data to be used by the models

Return type Tuple of DataTypes

ml_tooling.data.**load_demo_dataset**(*dataset_name: str*, ***kwargs*) → ml_tooling.data.base_data.Dataset
Create a `Dataset` implementing the demo datasets from sklearn.datasets

Parameters

- **dataset_name** (*str*) – Name of the dataset to use. If 'openml' is passed either parameter name or data_id needs to be specified.

  One of:

  - iris
  - boston
  - diabetes
  - digits
  - linnerud
  - wine
  - breast_cancer

– openml

- ****kwargs** – Kwargs are passed on to the scikit-learn dataset function

Returns An instance of *Dataset*

Return type *Dataset*

Dataset Plotting Methods

class ml_tooling.plots.viz.data_viz.**DataVisualize**(*data*)

missing_data(*ax: Optional[matplotlib.axes._axes.Axes] = None*, *top_n: Union[int, float, None] = None*, *bottom_n: Union[int, float, None] = None*, *feature_pipeline: Optional[sklearn.pipeline.Pipeline] = None*) → matplotlib.axes._axes.Axes

Plot number of missing data points per column. Sorted by number of missing values.

Also allows for selecting top_n/bottom_n number or percent of columns by passing an int or float

Parameters

- **ax** (*plt.Axes*) – Matplotlib axes to draw the graph on. Creates a new one by default
- **top_n** (*int, float*) – If top_n is an integer, return top_n features. If top_n is a float between (0, 1), return top_n percent features
- **bottom_n** (*int, float*) – If bottom_n is an integer, return bottom_n features. If bottom_n is a float between (0, 1), return bottom_n percent features
- **feature_pipeline** (Pipeline) – A feature transformation pipeline to be applied before graphing the final results

Returns

Return type plt.Axes

target_correlation(*method: str = 'spearman'*, *ax: Optional[matplotlib.axes._axes.Axes] = None*, *top_n: Union[int, float, None] = None*, *bottom_n: Union[int, float, None] = None*, *feature_pipeline: Optional[sklearn.pipeline.Pipeline] = None*) → matplotlib.axes._axes.Axes

Plot the correlation between each feature and the target variable using the given value.

Also allows selecting how many features to show by setting the top_n and/or bottom_n parameters.

Parameters

- **method** (*str*) – Which method to use when calculating correlation. Supports one of 'pearson', 'spearman', 'kendall'.
- **ax** (*plt.Axes*) – Matplotlib axes to draw the graph on. Creates a new one by default
- **top_n** (*int, float*) – If top_n is an integer, return top_n features. If top_n is a float between (0, 1), return top_n percent features
- **bottom_n** (*int, float*) – If bottom_n is an integer, return bottom_n features. If bottom_n is a float between (0, 1), return bottom_n percent features
- **feature_pipeline** (Pipeline) – A feature transformation pipeline to be applied before graphing the data

Returns

Return type plt.Axes

target_feature_distribution(*feature_name: str, method: str = 'mean', ax: Optional[matplotlib.axes._axes.Axes] = None, feature_pipeline: Optional[sklearn.pipeline.Pipeline] = None*) → matplotlib.axes._axes.Axes

Creates a plot which compares the mean or median of a binary target based on the given category feature.

Parameters

- **feature_name** (*str*) – Which feature showcase in plot.
- **method** (*str*) – Which method to compare with. One of 'median' or 'mean'.
- **ax** (*plt.Axes*) – Matplotlib axes to draw the graph on. Creates a new one by default
- **feature_pipeline** (Pipeline) – A feature transformation pipeline to be applied before graphing the data

Returns

Return type plt.Axes

Storage

class ml_tooling.storage.**FileStorage**(*dir_path: Union[str, pathlib.Path] = None*)

File Storage class for handling storage of estimators to the file system

get_list() → List[pathlib.Path]

Finds a list of estimator file paths in the FileStorage directory.

Example

Find and return estimator paths in a given directory: my_estimators = FileStorage('path/to/dir').get_list()

Returns list of paths to files sorted by filename

Return type List[Path]

load(*file_path: Union[str, pathlib.Path]*) → Any

Loads a joblib pickled estimator from given filepath and returns the unpickled object

Parameters **file_path** (*Pathlike*) – Path where to load the estimator file relative to FileStorage

Example

We can load a saved pickled estimator from disk directly from FileStorage:

storage = FileStorage('path/to/dir') my_estimator = storage.load('mymodel.pkl')

We now have a trained estimator loaded.

Returns The object loaded from disk

Return type Object

save(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], filename: str, prod: bool = False*) → pathlib.Path

Save a joblib pickled estimator.

Parameters

- **estimator** (*obj*) – The estimator object
- **filename** (*str*) – filename of estimator pickle file
- **prod** (*bool*) – Whether or not to save in "production mode" - Production mode saves to /src/<projectname>/ regardless of what FileStorage was instantiated with

Example

To save your trained estimator, use the FileStorage context manager.

storage = FileStorage('/path/to/save/dir/') file_path = storage.save(estimator, 'filename')

We now have saved an estimator to a pickle file.

Returns Path to the saved object

Return type Path

class ml_tooling.storage.**Storage**

Base class for Storage classes

get_list() → List[pathlib.Path]

Abstract method to be implemented by the user. Defines method used to show which objects have been saved

Returns Paths to each of the estimators sorted lexically

Return type List[Path]

load(*file_path: Union[str, pathlib.Path]*) → Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline]

Abstract method to be implemented by the user. Defines method used to load data from the storage type

Returns Returns the unpickled object

Return type Estimator

save(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], file_path: Union[str, pathlib.Path], prod: bool = False*) → Union[str, pathlib.Path]

Abstract method to be implemented by the user. Defines method used to save data from the storage type

Returns Path to where the pickled object is saved

Return type Pathlike

class ml_tooling.storage.**ArtifactoryStorage**(*artifactory_url: str, repo: str, apikey: Optional[str] = None, auth: Optional[Tuple[str, str]] = None*)

Artifactory Storage class for handling storage of estimators to JFrog artifactory

Example

Instantiate this class with a url and path to the repo like so:

storage = ArtifactoryStorage('http://artifactory.com','path/to/artifact')

get_list() → List[ArtifactoryPath]

Finds a list of estimator artifact paths in the ArtifactoryStorage repo.

Example

Find and return estimator paths in a given directory: my_estimators = ArtifactoryStorage('http://artifactory.com', 'path/to/repo').get_list()

Returns list of paths to files sorted by filename

Return type List[ArtifactoryPath]

load(*file_path: Union[str, pathlib.Path]*) → Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline]
Loads a pickled estimator from given filepath and returns the estimator

Parameters **file_path** (*Pathlike*) – Path to load the estimator relative to ArtifactoryStorage

Example

We can load a saved pickled estimator from disk directly from Artifactory:

storage = ArtifactoryStorage('http://artifactory.com', 'path/to/repo') my_estimator = storage.load('estimatorfile')

We now have a trained estimator loaded.

Returns estimator unpickled object

Return type Object

save(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], filename: str, prod: bool = False*) → ArtifactoryPath
Save a pickled estimator to artifactory.

Parameters

- **estimator** (*Estimator*) – The estimator object
- **filename** (*str*) – filename of estimator pickle file
- **prod** (*bool*) – Production variable, set to True if saving a production-ready estimator

Example

To save your trained estimator:

storage = ArtifactoryStorage('http://artifactory.com', 'path/to/repo') artifactory_path = storage.save(estimator, 'estimator.pkl')

We now have saved an estimator to a pickle file.

Returns File path to stored estimator

Return type ArtifactoryPath

Config

All configuration options available

class ml_tooling.config.**DefaultConfig**
Configuration for Models

VERBOSITY = 0 The level of verbosity from output

CLASSIFIER_METRIC = 'accuracy' Default metric for classifiers

REGRESSION_METRIC = 'r2' Default metric for regressions

CROSS_VALIDATION = 10 Default Number of cross validation folds to use

N_JOBS = -1 Default number of cores to use when doing multiprocessing. -1 means use all available

RANDOM_STATE = 42 Default random state seed for all functions involving randomness

RUN_DIR = './runs' Default folder to store run logging files

ESTIMATOR_DIR = './models' Default folder to store pickled models in

LOG = False Toggles whether or not to log runs to a file. Set to True if you want every run to be logged, else use the `log()` context manager

TRAIN_TEST_SHUFFLE = True Default whether or not to shuffle data for test set

TEST_SIZE = 0.25 Default percentage of data that will be part of the test set

Result

Result class to work with results from scoring a model

class ml_tooling.result.**Result**(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], metrics: ml_tooling.metrics.metric.Metrics, data: ml_tooling.data.base_data.Dataset*)

Contains the result of a given training run. Contains plotting methods, as well as being comparable with other results

Parameters

- **estimator** (*Estimator*) – Estimator used to generate the result
- **metrics** (Metrics) – Metrics used to score the model
- **data** (Dataset) – Dataset used to generate the result

Method generated by attrs for class Result.

classmethod from_estimator(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], data: ml_tooling.data.base_data.Dataset, metrics: List[str], cv: Any = None, n_jobs: int = None, verbose: int = 0*) → ml_tooling.result.result.Result

Create a result from an estimator

Parameters

- **estimator** (*Estimator*) – The trained estimator
- **data** (Dataset) – The dataset used to create the result
- **metrics** (*List[str]*) – What metrics to calculate
- **cv** (*Any*) – If CV is an int, run with that many CV iterations. If a sklearn-compatible CV object is passed, use that object to generate the CV splits
- **n_jobs** (*int*) – How many jobs to parallelize with
- **verbose** (*int*) – How verbose should the parallelization be

Returns An instance of Result

Return type *Result*

log(*saved_estimator_path=None*, *savedir=None*) → ml_tooling.logging.log_estimator.Log
Generate a Log object from the result

Parameters

- **saved_estimator_path** (*Union[str, pathlib.Path, None]*) – Optionally, where the saved estimator is
- **savedir** (*Union[str, pathlib.Path, None]*) – An optional path of where to save the log

Returns An instance of Log

Return type Log

model
Return the model used in the result

parameters
Return the estimator params

ResultGroup

A container of Results - some methods in ML Tooling return multiple results, which will be grouped into a ResultGroup. A ResultGroup is sorted by the Result metric and proxies attributes to the best result

class ml_tooling.result.**ResultGroup**(*results: List[ml_tooling.result.result.Result]*)
A container for results. Proxies attributes to the best result. Supports indexing like a list.

Method generated by attrs for class ResultGroup.

Classification Result Visualizations

class ml_tooling.plots.viz.**ClassificationVisualize**(*estimator*, *data*)
Visualization class for Classification models

confusion_matrix(*normalized: bool = True*, *threshold: Optional[float] = None*, ***kwargs*) → matplotlib.axes._axes.Axes
Visualize a confusion matrix for a classification estimator Any kwargs are passed onto matplotlib

Parameters

- **normalized** (*bool*) – Whether or not to normalize annotated class counts
- **threshold** (*float*) – Threshold to use for classification - defaults to 0.5

Returns Returns a Confusion Matrix plot

Return type plt.Axes

default_metric
Finds estimator_type for estimator in a BaseVisualize and returns default metric for this class stated in .config. If passed estimator is a Pipeline, assume last step is the estimator.

Returns Name of the metric

Return type str

feature_importance(*top_n: Union[int, float] = None*, *bottom_n: Union[int, float] = None*, *class_index: int = None*, *add_label: bool = True*, *ax: matplotlib.axes._axes.Axes = None*, ***kwargs*) → matplotlib.axes._axes.Axes

Visualizes feature importance of the estimator through permutation.

Parameters

- **top_n** (*int, float*) – If top_n is an integer, return top_n features. If top_n is a float between (0, 1), return top_n percent features
- **bottom_n** (*int, float*) – If bottom_n is an integer, return bottom_n features. If bottom_n is a float between (0, 1), return bottom_n percent features
- **class_index** (*int, optional*) – In a multi-class setting, plot the feature importances for the given label. If None, assume a binary classification
- **add_label** (*bool*) – Toggles value labels on end of each bar
- **ax** (*Axes*) – Draws graph on passed ax - otherwise creates new ax
- **kwargs** (*dict*) – Passed to plt.barh

Returns

Return type matplotlib.Axes

learning_curve(*cv: int = None*, *scoring: str = 'default'*, *n_jobs: int = None*, *train_sizes: Sequence[float] = array([0.1, 0.325, 0.55, 0.775, 1.])*, *ax: matplotlib.axes._axes.Axes = None*, ***kwargs*) → matplotlib.axes._axes.Axes

Generates a `learning_curve()` plot, used to determine model performance as a function of number of training examples.

Illustrates whether or not number of training examples is the performance bottleneck. Also used to diagnose underfitting or overfitting, by seeing how the training set and validation set performance differ.

Parameters

- **cv** (*int*) – Number of CV iterations to run
- **scoring** (*str*) – Metric to use in scoring - must be a scikit-learn compatible scoring method
- **n_jobs** (*int*) – Number of jobs to use in parallelizing the estimator fitting and scoring
- **train_sizes** (*Sequence of floats*) – Percentage intervals of data to use when training
- **ax** (*plt.Axes*) – The plot will be drawn on the passed ax - otherwise a new figure and ax will be created.
- **kwargs** (*dict*) – Passed along to matplotlib line plots

Returns

Return type plt.Axes

lift_curve(***kwargs*) → matplotlib.axes._axes.Axes

Visualize a Lift Curve for a classification estimator Estimator must implement a *predict_proba* method Any kwargs are passed onto matplotlib

Parameters **kwargs** (*optional*) – Keyword arguments to pass on to matplotlib

Returns

Return type plt.Axes

permutation_importance(*n_repeats: int = 5*, *scoring: str = 'default'*, *top_n: Union[int, float] = None*, *bottom_n: Union[int, float] = None*, *add_label: bool = True*, *n_jobs: int = None*, *ax: matplotlib.axes._axes.Axes = None*, ***kwargs*) → matplotlib.axes._axes.Axes

Visualizes feature importance of the estimator through permutation.

Parameters

- **n_repeats** (*int*) – Number of times to permute a feature
- **scoring** (*str*) – Metric to use in scoring - must be a scikit-learn compatible scoring method
- **top_n** (*int, float*) – If top_n is an integer, return top_n features. If top_n is a float between (0, 1), return top_n percent features
- **bottom_n** (*int, float*) – If bottom_n is an integer, return bottom_n features. If bottom_n is a float between (0, 1), return bottom_n percent features
- **add_label** (*bool*) – Toggles value labels on end of each bar
- **ax** (*Axes*) – Draws graph on passed ax - otherwise creates new ax
- **n_jobs** (*int, optional*) – Number of parallel jobs to run. Defaults to N_JOBS setting in config.
- **kwargs** (*dict*) – Passed to plt.barh

Returns

Return type matplotlib.Axes

precision_recall_curve(*labels: List[str] = None*, ***kwargs*) → matplotlib.axes._axes.Axes

Visualize a Precision-Recall curve for a classification estimator. Estimator must implement a *predict_proba* method. Any kwargs are passed onto matplotlib.

Parameters

- **labels** (*List of str*) – Labels to use for the class names if multi-class
- **kwargs** (*optional*) – Keyword arguments to pass on to matplotlib

Returns Plot of precision-recall curve

Return type plt.Axes

roc_curve(*labels: List[str] = None*, ***kwargs*) → matplotlib.axes._axes.Axes

Visualize a ROC curve for a classification estimator. Estimator must implement a *predict_proba* method Any kwargs are passed onto matplotlib

Parameters

- **labels** (*List of str*) – Labels to use for the class names if multi-class
- **kwargs** (*optional*) – Keyword arguments to pass on to matplotlib

Returns Returns a ROC AUC plot

Return type plt.Axes

validation_curve(*param_name: str, param_range: Sequence[T_co], n_jobs: int = None, cv: int = None, scoring: str = 'default', ax: matplotlib.axes._axes.Axes = None, **kwargs*) → matplotlib.axes._axes.Axes

Generates a `validation_curve()` plot, graphing the impact of changing a hyperparameter on the scoring metric.

This lets us examine how a hyperparameter affects over/underfitting by examining train/test performance with different values of the hyperparameter.

Parameters

- **param_name** (*str*) – Name of hyperparameter to plot
- **param_range** (*Sequence*) – The individual values to plot for *param_name*
- **n_jobs** (*int*) – Number of jobs to use in parallelizing the estimator fitting and scoring
- **cv** (*int*) – Number of CV iterations to run. Defaults to value in *Model.config*. Uses a StratifiedKFold if'estimator' is a classifier - otherwise a KFold is used.
- **scoring** (*str*) – Metric to use in scoring - must be a scikit-learn compatible scoring method
- **ax** (*plt.Axes*) – The plot will be drawn on the passed ax - otherwise a new figure and ax will be created.
- **kwargs** (*dict*) – Passed along to matplotlib line plots

Returns

Return type plt.Axes

Regression Result Visualizations

class ml_tooling.plots.viz.**RegressionVisualize**(*estimator*, *data*)

Visualization class for Regression models

default_metric

Finds estimator_type for estimator in a BaseVisualize and returns default metric for this class stated in .config. If passed estimator is a Pipeline, assume last step is the estimator.

Returns Name of the metric

Return type str

feature_importance(*top_n: Union[int, float] = None*, *bottom_n: Union[int, float] = None*, *class_index: int = None*, *add_label: bool = True*, *ax: matplotlib.axes._axes.Axes = None*, ***kwargs*) → matplotlib.axes._axes.Axes

Visualizes feature importance of the estimator through permutation.

Parameters

- **top_n** (*int, float*) – If top_n is an integer, return top_n features. If top_n is a float between (0, 1), return top_n percent features
- **bottom_n** (*int, float*) – If bottom_n is an integer, return bottom_n features. If bottom_n is a float between (0, 1), return bottom_n percent features
- **class_index** (*int, optional*) – In a multi-class setting, plot the feature importances for the given label. If None, assume a binary classification
- **add_label** (*bool*) – Toggles value labels on end of each bar
- **ax** (*Axes*) – Draws graph on passed ax - otherwise creates new ax
- **kwargs** (*dict*) – Passed to plt.barh

Returns

Return type matplotlib.Axes

learning_curve(*cv: int = None*, *scoring: str = 'default'*, *n_jobs: int = None*, *train_sizes: Sequence[float] = array([0.1, 0.325, 0.55, 0.775, 1.])*, *ax: matplotlib.axes._axes.Axes = None*, ***kwargs*) → matplotlib.axes._axes.Axes

Generates a `learning_curve()` plot, used to determine model performance as a function of number of training examples.

Illustrates whether or not number of training examples is the performance bottleneck. Also used to diagnose underfitting or overfitting, by seeing how the training set and validation set performance differ.

Parameters

- **cv** (*int*) – Number of CV iterations to run
- **scoring** (*str*) – Metric to use in scoring - must be a scikit-learn compatible scoring method
- **n_jobs** (*int*) – Number of jobs to use in parallelizing the estimator fitting and scoring
- **train_sizes** (*Sequence of floats*) – Percentage intervals of data to use when training
- **ax** (*plt.Axes*) – The plot will be drawn on the passed ax - otherwise a new figure and ax will be created.
- **kwargs** (*dict*) – Passed along to matplotlib line plots

Returns

Return type plt.Axes

permutation_importance(*n_repeats: int = 5*, *scoring: str = 'default'*, *top_n: Union[int, float] = None*, *bottom_n: Union[int, float] = None*, *add_label: bool = True*, *n_jobs: int = None*, *ax: matplotlib.axes._axes.Axes = None*, ***kwargs*) → matplotlib.axes._axes.Axes

Visualizes feature importance of the estimator through permutation.

Parameters

- **n_repeats** (*int*) – Number of times to permute a feature
- **scoring** (*str*) – Metric to use in scoring - must be a scikit-learn compatible scoring method
- **top_n** (*int, float*) – If top_n is an integer, return top_n features. If top_n is a float between (0, 1), return top_n percent features
- **bottom_n** (*int, float*) – If bottom_n is an integer, return bottom_n features. If bottom_n is a float between (0, 1), return bottom_n percent features
- **add_label** (*bool*) – Toggles value labels on end of each bar
- **ax** (*Axes*) – Draws graph on passed ax - otherwise creates new ax
- **n_jobs** (*int, optional*) – Number of parallel jobs to run. Defaults to N_JOBS setting in config.
- **kwargs** (*dict*) – Passed to plt.barh

Returns

Return type matplotlib.Axes

prediction_error(***kwargs*) → matplotlib.axes._axes.Axes

Visualizes prediction error of a regression estimator Any kwargs are passed onto matplotlib

Returns Plot of the estimator's prediction error

Return type matplotlib.Axes

residuals(***kwargs*) → matplotlib.axes._axes.Axes
Visualizes residuals of a regression estimator. Any kwargs are passed onto matplotlib

Returns Plot of the estimator's residuals

Return type matplotlib.Axes

validation_curve(*param_name: str, param_range: Sequence[T_co], n_jobs: int = None, cv: int = None, scoring: str = 'default', ax: matplotlib.axes._axes.Axes = None, **kwargs*) → matplotlib.axes._axes.Axes
Generates a `validation_curve()` plot, graphing the impact of changing a hyperparameter on the scoring metric.

This lets us examine how a hyperparameter affects over/underfitting by examining train/test performance with different values of the hyperparameter.

Parameters

- **param_name** (*str*) – Name of hyperparameter to plot
- **param_range** (*Sequence*) – The individual values to plot for *param_name*
- **n_jobs** (*int*) – Number of jobs to use in parallelizing the estimator fitting and scoring
- **cv** (*int*) – Number of CV iterations to run. Defaults to value in *Model.config*. Uses a `StratifiedKFold` if'estimator' is a classifier - otherwise a `KFold` is used.
- **scoring** (*str*) – Metric to use in scoring - must be a scikit-learn compatible scoring method
- **ax** (*plt.Axes*) – The plot will be drawn on the passed ax - otherwise a new figure and ax will be created.
- **kwargs** (*dict*) – Passed along to matplotlib line plots

Returns

Return type plt.Axes

Plots

ml_tooling.plots.**plot_confusion_matrix**(*y_true: Union[pandas.core.series.Series, numpy.ndarray], y_pred: Union[pandas.core.series.Series, numpy.ndarray], normalized: bool = True, title: str = None, ax: matplotlib.axes._axes.Axes = None, labels: Sequence[str] = None*) → matplotlib.axes._axes.Axes
Plots a confusion matrix of predicted labels vs actual labels

Parameters

- **y_true** – True labels
- **y_pred** – Predicted labels from estimator
- **normalized** – Whether to normalize counts in matrix
- **title** – Title for plot
- **ax** – Pass your own ax
- **labels** – Pass custom list of labels

Returns matplotlib.Axes

ml_tooling.plots.**plot_target_correlation**(*features: pandas.core.frame.DataFrame, target: Union[pandas.core.series.Series, numpy.ndarray], method: str = 'spearman', ax: matplotlib.axes._axes.Axes = None, top_n: Union[int, float] = None, bottom_n: Union[int, float] = None, title: str = 'Feature-Target Correlation'*) → matplotlib.axes._axes.Axes

Plot the correlation between each feature and the target variable using the given value.

Also allows selecting how many features to show by setting the top_n and/or bottom_n parameters.

Parameters

- **features** (*pd.DataFrame*) – Features to plot
- **target** (*np.Array or pd.Series*) – Target to calculate correlation with
- **method** (*str*) – Which method to use when calculating correlation. Supports one of 'pearson', 'spearman', 'kendall'.
- **ax** (*plt.Axes*) – Matplotlib axes to draw the graph on. Creates a new one by default
- **top_n** (*int, float*) – If top_n is an integer, return top_n features. If top_n is a float between (0, 1), return top_n percent features
- **bottom_n** (*int, float*) – If bottom_n is an integer, return bottom_n features. If bottom_n is a float between (0, 1), return bottom_n percent features
- **title** (*str*) – Title of graph

Returns

Return type plt.Axes

ml_tooling.plots.**plot_feature_importance**(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], x: pandas.core.frame.DataFrame, ax: matplotlib.axes._axes.Axes = None, class_index: int = None, bottom_n: Union[int, float] = None, top_n: Union[int, float] = None, add_label: bool = True, title: str = '', **kwargs*) → matplotlib.axes._axes.Axes

Plot either the estimator coefficients or the estimator feature importances depending on what is provided by the estimator.

see also :func:ml_tooling.plot.plot_permutation_importance for an unbiased version of feature importance using permutation importance

Parameters

- **estimator** (*Estimator*) – Estimator to use to calculate permuted feature importance
- **x** (*DataType*) – Features to calculate permuted feature importance for
- **ax** (*Axes*) – Matplotlib axes to draw the graph on. Creates a new one by default
- **class_index** (*int, optional*) – In a multi-class setting, choose which class to get feature importances for. If None, will assume a binary classifier
- **bottom_n** (*int*) – Plot only bottom n features
- **top_n** (*int*) – Plot only top n features

- **add_label** (*bool*) – Whether or not to plot text labels for the bars
- **title** (*str*) – Title to add to the plot
- **kwargs** (*dict*) – Any kwargs are passed to matplotlib

Returns

Return type plt.Axes

ml_tooling.plots.**plot_lift_curve**(*y_true: Union[pandas.core.series.Series, numpy.ndarray], y_proba: Union[pandas.core.series.Series, numpy.ndarray], title: str = None, ax: matplotlib.axes._axes.Axes = None, labels: List[str] = None, threshold: float = 0.5*) → matplotlib.axes._axes.Axes

Plot a lift chart from results. Also calculates lift score based on a .5 threshold

Parameters

- **y_true** (*DataType*) – True labels
- **y_proba** (*DataType*) – Model's predicted probability
- **title** (*str*) – Plot title
- **ax** (*Axes*) – Pass your own ax
- **labels** (*List of str*) – Labels to use per class
- **threshold** (*float*) – Threshold to use when determining lift score

Returns

Return type matplotlib.Axes

ml_tooling.plots.**plot_prediction_error**(*y_true: Union[pandas.core.series.Series, numpy.ndarray], y_pred: Union[pandas.core.series.Series, numpy.ndarray], title: str = None, ax: matplotlib.axes._axes.Axes = None*) → matplotlib.axes._axes.Axes

Plots prediction error of regression estimator

Parameters

- **y_true** – True values
- **y_pred** – Model's predicted values
- **title** – Plot title
- **ax** – Pass your own ax

Returns matplotlib.Axes

ml_tooling.plots.**plot_residuals**(*y_true: Union[pandas.core.series.Series, numpy.ndarray], y_pred: Union[pandas.core.series.Series, numpy.ndarray], title: str = None, ax: matplotlib.axes._axes.Axes = None*) → matplotlib.axes._axes.Axes

Plots residuals from a regression.

Parameters

- **y_true** – True values
- **y_pred** – Models predicted value
- **title** – Plot title

- **ax** – Pass your own ax

Returns matplotlib.Axes

ml_tooling.plots.**plot_roc_auc**(*y_true: Union[pandas.core.series.Series, numpy.ndarray], y_proba: Union[pandas.core.series.Series, numpy.ndarray], title: str = None, ax: matplotlib.axes._axes.Axes = None, labels: List[str] = None*) → matplotlib.axes._axes.Axes

Plot ROC AUC curve. Works only with probabilities

Parameters

- **y_true** (*DataType*) – True labels
- **y_proba** (*DataType*) – Probability estimate from estimator
- **title** (*str*) – Plot title
- **ax** (*Axes*) – Pass in your own ax
- **labels** (*List of str*) – Optionally specify label names

Returns Plot of ROC AUC curve

Return type plt.Axes

ml_tooling.plots.**plot_pr_curve**(*y_true: Union[pandas.core.series.Series, numpy.ndarray], y_proba: Union[pandas.core.series.Series, numpy.ndarray], title: str = None, ax: matplotlib.axes._axes.Axes = None, labels: List[str] = None*) → matplotlib.axes._axes.Axes

Plot precision-recall curve. Works only with probabilities.

Parameters

- **y_true** (*DataType*) – True labels
- **y_proba** (*DataType*) – Probability estimate from estimator
- **title** (*str*) – Plot title
- **ax** (*plt.Axes*) – Pass in your own ax
- **labels** (*List of str, optional*) – Labels for each class

Returns Plot of precision-recall curve

Return type plt.Axes

ml_tooling.plots.**plot_learning_curve**(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], x: pandas.core.frame.DataFrame, y: Union[pandas.core.series.Series, numpy.ndarray], cv: int = 5, scoring: str = 'default', n_jobs: int = -1, train_sizes: Sequence[T_co] = array([0.1 , 0.325, 0.55 , 0.775, 1.]), ax: matplotlib.axes._axes.Axes = None, random_state: int = None, title: str = 'Learning Curve', **kwargs*) → matplotlib.axes._axes.Axes

Generates a `learning_curve()` plot, used to determine model performance as a function of number of training examples.

Illustrates whether or not number of training examples is the performance bottleneck. Also used to diagnose underfitting or overfitting, by seeing how the training set and validation set performance differ.

Parameters

- **estimator** (*sklearn-compatible estimator*) – An instance of a sklearn estimator
- **x** (*pd.DataFrame*) – DataFrame of features
- **y** (*pd.Series or np.Array*) – Target values to predict
- **cv** (*int*) – Number of CV iterations to run. Uses a `StratifiedKFold` if *estimator* is a classifier - otherwise a `KFold` is used.
- **scoring** (*str*) – Metric to use in scoring - must be a scikit-learn compatible scoring method
- **n_jobs** (*int*) – Number of jobs to use in parallelizing the estimator fitting and scoring
- **train_sizes** (*Sequence of floats*) – Percentage intervals of data to use when training
- **ax** (*plt.Axes*) – The plot will be drawn on the passed ax - otherwise a new figure and ax will be created.
- **random_state** (*int*) – Random state to use in CV splitting
- **title** (*str*) – Title to be used on the plot
- **kwargs** (*dict*) – Passed along to matplotlib line plots

Returns

Return type plt.Axes

ml_tooling.plots.**plot_validation_curve**(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], x: pandas.core.frame.DataFrame, y: Union[pandas.core.series.Series, numpy.ndarray], param_name: str, param_range: Sequence[T_co], cv: int = 5, scoring: str = 'default', n_jobs: int = -1, ax: matplotlib.axes._axes.Axes = None, title: str = '', **kwargs*) → matplotlib.axes._axes.Axes

Plots a `validation_curve()`, graphing the impact of changing a hyperparameter on the scoring metric.

This lets us examine how a hyperparameter affects over/underfitting by examining train/test performance with different values of the hyperparameter.

Parameters

- **estimator** (*sklearn-compatible estimator*) – An instance of a sklearn estimator
- **x** (*pd.DataFrame*) – DataFrame of features
- **y** (*pd.Series or np.Array*) – Target values to predict
- **param_name** (*str*) – Name of hyperparameter to plot
- **param_range** (*Sequence*) – The individual values to plot for *param_name*
- **cv** (*int*) – Number of CV iterations to run. Uses a `StratifiedKFold` if *estimator* is a classifier - otherwise a `KFold` is used.
- **scoring** (*str*) – Metric to use in scoring - must be a scikit-learn compatible scoring method
- **n_jobs** (*int*) – Number of jobs to use in parallelizing the estimator fitting and scoring

- **ax** (*plt.Axes*) – The plot will be drawn on the passed ax - otherwise a new figure and ax will be created.
- **title** (*str*) – Title to be used on the plot
- **kwargs** (*dict*) – Passed along to matplotlib line plots

Returns

Return type plt.Axes

ml_tooling.plots.**plot_missing_data**(*df: pandas.core.frame.DataFrame*, *ax: Optional[matplotlib.axes._axes.Axes] = None*, *top_n: Union[int, float, None] = None*, *bottom_n: Union[int, float, None] = None*, ***kwargs*) → matplotlib.axes._axes.Axes

Plot number of missing data points per column. Sorted by number of missing values.

Also allows for selecting top_n/bottom_n number or percent of columns by passing an int or float

Parameters

- **df** (*pd.DataFrame*) – Feature DataFrame to calculate missing values from
- **ax** (*plt.Axes*) – Matplotlib axes to draw the graph on. Creates a new one by default
- **top_n** (*int, float*) – If top_n is an integer, return top_n features. If top_n is a float between (0, 1), return top_n percent features
- **bottom_n** (*int, float*) – If bottom_n is an integer, return bottom_n features. If bottom_n is a float between (0, 1), return bottom_n percent features

Returns

Return type plt.Axes

ml_tooling.plots.**plot_target_feature_distribution**(*target: Union[pandas.core.series.Series, numpy.ndarray], feature: Union[pandas.core.series.Series, numpy.ndarray], title: str = 'Target feature distribution', method: str = 'mean', ax: matplotlib.axes._axes.Axes = None*) → matplotlib.axes._axes.Axes

Creates a plot which compares the mean or median of a binary target based on the given category features. NaN values are ignored in the calculation.

Parameters

- **target** (*DataType*) – Target to aggregate per feature category
- **feature** (*DataType*) – Categorical feature to group by
- **title** (*str*) – Title of graph
- **method** (*str*) – Which method to compare with. One of 'median' or 'mean'.
- **ax** (*plt.Axes*) – Matplotlib axes to draw the graph on. Creates a new one by default

Returns

Return type plt.Axes

Transformers

class ml_tooling.transformers.**Binarize**(*value: Any = None*)
Sets all instances of value to 1 and all others to 0 Returns a pandas DataFrame

Parameters **value** (*Any*) – The value to be set to 1

class ml_tooling.transformers.**Binner**(*bins: Union[int, list] = 5*, *labels: list = None*)
Bins data according to passed bins and labels. Uses `pandas.cut()` under the hood, see for further details

Parameters

- **bins** (*int, list*) – The criteria to bin by. An int value defines the number of equal-width bins in the range of x. The range of x is extended by .1% on each side to include the minimum and maximum values of x. If a list is passed, defines the bin edges allowing for non-uniform width and no extension of the range of x is done.
- **labels** (*list*) – Specifies the labels for the returned bins. Must be the same length as the resulting bins.

class ml_tooling.transformers.**ToCategorical**
Converts a column into a one-hot encoded column through pd.Categorical

class ml_tooling.transformers.**DateEncoder**(*day: bool = True*, *month: bool = True*, *week: bool = True*, *year: bool = True*)
Converts a date column into multiple day-month-year columns

Parameters

- **day** (*bool*) – If True, a new day column will be added.
- **month** (*bool*) – If True, a new month column will be added.
- **week** (*bool*) – If True, a new week column will be added.
- **year** (*bool*) – If True, a new year column will be added.

class ml_tooling.transformers.**DFFeatureUnion**(*transformer_list: list*)
Merges together two pipelines based on index.

Parameters **transformer_list** (*list*) – transformer_list is a list of (*name*, *transformer*) tuples, where *transfomer* implements fit/transform.

class ml_tooling.transformers.**FillNA**(*value: Union[str, int, None] = None*, *strategy: Optional[str] = None*, *indicate_nan: bool = False*)
Fills NA values with given value or strategy. Either a value or a strategy must be passed.

Parameters

- **value** (*str, int*) – A specific value to replace NaNs with.
- **strategy** (*str*) – A named strategy to replace NaNs with. One of 'mean', 'median', 'most_freq', 'max', 'min'
- **indicate_nan** (*bool*) – If True, a new column is added which indicates if a value in a column was missing.

class ml_tooling.transformers.**FreqFeature**
Converts a column into its normalized value count

class ml_tooling.transformers.**FuncTransformer**(*func: Callable[[...], pandas.core.frame.DataFrame] = None*, ***kwargs*)
Applies a given function to each column

Parameters

- **func** (*Callable[.., pd.DataFrame]*) – Define the function which should be applied on each column.
- **kwargs** – Specific for the selected func.

class ml_tooling.transformers.**DFRowFunc**(*strategy: Union[Callable[[...], pandas.core.frame.DataFrame], str] = None*)

Row-wise operation on Pandas DataFrame.

Parameters **strategy** (*Callable[.., pd.DataFrame], str*) – Strategy can either be one of the predefined or a callable. If some elements in the row are NaN these elements are ignored for the built-in strategies. Valid strategies are:

- sum
- min
- max
- mean

If a callable is used, it must return a pd.Series

class ml_tooling.transformers.**RareFeatureEncoder**(*threshold: Union[int, float] = 0.2, fill_rare: Any = 'Rare'*)

Replaces categories with a specified value, if they occur less often than the provided threshold.

Parameters

- **threshold** (*int, float*) – Sets the threshold for when a value is considered rare. Any value which occurs less than the threshold will be replaced with fill_rare. If threshold is a float, it will be considered a percentage and if it is an int, threshold will be considered the minimum number of observations.
- **fill_rare** (*Any*) – Fill value to use when replacing rare categories.

class ml_tooling.transformers.**Renamer**(*column_names: Union[list, str] = None*)

Renames columns to passed names.

Parameters **column_names** (*list, str*) – The column names which should replace the original column names.

class ml_tooling.transformers.**DFStandardScaler**(*copy: bool = True, with_mean: bool = True, with_std: bool = True*)

Wrapping of the StandardScaler from scikit-learn for Pandas DataFrames. See: StandardScaler

Parameters

- **copy** (*bool*) – If True, a copy of the dataframe is made.
- **with_mean** (*bool*) – If True, center the data before scaling.
- **with_std** (*bool*) – If True, scale the data to unit standard deviation.

class ml_tooling.transformers.**Select**(*columns: Union[List[str], str] = None*)

Selects columns from DataFrame

Parameters **columns** (*List[str], str, None*) – Specify which columns are selected.

class ml_tooling.transformers.**Pipeline**(*steps*, *, *memory=None*, *verbose=False*)

Pipeline of transforms with a final estimator.

Sequentially apply a list of transforms and a final estimator. Intermediate steps of the pipeline must be 'transforms', that is, they must implement fit and transform methods. The final estimator only needs to implement fit. The transformers in the pipeline can be cached using `memory` argument.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a '__', as in the example below. A step's estimator may be replaced entirely by setting the parameter with its name to another estimator, or a transformer removed by setting it to 'passthrough' or `None`.

Read more in the User Guide.

New in version 0.5.

Parameters

- **steps** (*list*) – List of (name, transform) tuples (implementing fit/transform) that are chained, in the order in which they are chained, with the last object an estimator.
- **memory** (*str or object with the joblib.Memory interface, default=None*) – Used to cache the fitted transformers of the pipeline. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.
- **verbose** (*bool, default=False*) – If True, the time elapsed while fitting each step will be printed as it is completed.

named_steps
Dictionary-like object, with the following attributes. Read-only attribute to access any step parameter by user given name. Keys are step names and values are steps parameters.

Type `Bunch`

See also:

make_pipeline Convenience function for simplified pipeline construction.

Examples

```
>>> from sklearn.svm import SVC
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.pipeline import Pipeline
>>> X, y = make_classification(random_state=0)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
...                                                     random_state=0)
>>> pipe = Pipeline([('scaler', StandardScaler()), ('svc', SVC())])
>>> # The pipeline can be used as any other estimator
>>> # and avoids leaking the test set into the train set
>>> pipe.fit(X_train, y_train)
Pipeline(steps=[('scaler', StandardScaler()), ('svc', SVC())])
>>> pipe.score(X_test, y_test)
0.88
```

decision_function(*X*)
Apply transforms, and decision_function of the final estimator

Parameters **X** (*iterable*) – Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns **y_score**

Return type array-like of shape (n_samples, n_classes)

fit (*X*, *y=None*, ***fit_params*)
Fit the model

Fit all the transforms one after the other and transform the data, then fit the transformed data using the final estimator.

Parameters

- **X** (*iterable*) – Training data. Must fulfill input requirements of first step of the pipeline.
- **y** (*iterable, default=None*) – Training targets. Must fulfill label requirements for all steps of the pipeline.
- ****fit_params** (*dict of string -> object*) – Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

Returns **self** – This estimator

Return type *Pipeline*

fit_predict (*X*, *y=None*, ***fit_params*)
Applies fit_predict of last step in pipeline after transforms.

Applies fit_transforms of a pipeline to the data, followed by the fit_predict method of the final estimator in the pipeline. Valid only if the final estimator implements fit_predict.

Parameters

- **X** (*iterable*) – Training data. Must fulfill input requirements of first step of the pipeline.
- **y** (*iterable, default=None*) – Training targets. Must fulfill label requirements for all steps of the pipeline.
- ****fit_params** (*dict of string -> object*) – Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

Returns **y_pred**

Return type array-like

fit_transform (*X*, *y=None*, ***fit_params*)
Fit the model and transform with the final estimator

Fits all the transforms one after the other and transforms the data, then uses fit_transform on transformed data with the final estimator.

Parameters

- **X** (*iterable*) – Training data. Must fulfill input requirements of first step of the pipeline.
- **y** (*iterable, default=None*) – Training targets. Must fulfill label requirements for all steps of the pipeline.
- ****fit_params** (*dict of string -> object*) – Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

Returns **Xt** – Transformed samples

Return type array-like of shape (n_samples, n_transformed_features)

get_params(*deep=True*)
Get parameters for this estimator.

Returns the parameters given in the constructor as well as the estimators contained within the *steps* of the *Pipeline*.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type mapping of string to any

inverse_transform
Apply inverse transformations in reverse order

All estimators in the pipeline must support `inverse_transform`.

Parameters **Xt** (*array-like of shape (n_samples, n_transformed_features)*) – Data samples, where `n_samples` is the number of samples and `n_features` is the number of features. Must fulfill input requirements of last step of pipeline's `inverse_transform` method.

Returns **Xt**

Return type array-like of shape (n_samples, n_features)

predict(*X*, ***predict_params*)
Apply transforms to the data, and predict with the final estimator

Parameters

- **X** (*iterable*) – Data to predict on. Must fulfill input requirements of first step of the pipeline.
- ****predict_params** (*dict of string -> object*) – Parameters to the `predict` called at the end of all transformations in the pipeline. Note that while this may be used to return uncertainties from some models with return_std or return_cov, uncertainties that are generated by the transformations in the pipeline are not propagated to the final estimator.

  New in version 0.20.

Returns **y_pred**

Return type array-like

predict_log_proba(*X*)
Apply transforms, and predict_log_proba of the final estimator

Parameters **X** (*iterable*) – Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns **y_score**

Return type array-like of shape (n_samples, n_classes)

predict_proba(*X*)
Apply transforms, and predict_proba of the final estimator

Parameters **X** (*iterable*) – Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns y_proba

Return type array-like of shape (n_samples, n_classes)

score(*X*, *y=None*, *sample_weight=None*)
Apply transforms, and score with the final estimator

Parameters

- **X** (*iterable*) – Data to predict on. Must fulfill input requirements of first step of the pipeline.
- **y** (*iterable, default=None*) – Targets used for scoring. Must fulfill label requirements for all steps of the pipeline.
- **sample_weight** (*array-like, default=None*) – If not None, this argument is passed as `sample_weight` keyword argument to the `score` method of the final estimator.

Returns score

Return type float

score_samples(*X*)
Apply transforms, and score_samples of the final estimator.

Parameters **X** (*iterable*) – Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns y_score

Return type ndarray of shape (n_samples,)

set_params(***kwargs*)
Set the parameters of this estimator.

Valid parameter keys can be listed with `get_params()`. Note that you can directly set the parameters of the estimators contained in *steps*.

Returns

Return type self

transform
Apply transforms, and transform with the final estimator

This also works where final estimator is `None`: all prior transformations are applied.

Parameters **X** (*iterable*) – Data to transform. Must fulfill input requirements of first step of the pipeline.

Returns Xt

Return type array-like of shape (n_samples, n_transformed_features)

Metric

class ml_tooling.metrics.**Metric**(*name: str*, *score: float = None*, *cross_val_scores: Optional[numpy.ndarray] = None*)
Represents a single metric, containing a metric name and its corresponding score. Can be instantiated using any sklearn-compatible *score_* strings

A Metric knows how to generate it's own score by calling *score_metric()*, passing an estimator, an X and a Y. A Metric can also get a cross-validated score by calling *score_metric_cv()* and passing a CV value - either a *CV_* object or an int specifying number of folds

Examples

```
>>> from ml_tooling.metrics import Metric
>>> from sklearn.linear_model import LinearRegression
>>> import numpy as np
>>> metric = Metric('r2')
>>> x = np.array([[1],[2],[3],[4]])
>>> y = np.array([[2], [4], [6], [8]])
>>> estimator = LinearRegression().fit(x, y)
>>> metric.score_metric(estimator, x, y)
Metric(name='r2', score=1.0)
>>> metric.score
1.0
>>> metric.name
'r2'
```

```
>>> metric.score_metric_cv(estimator, x, y, cv=2)
Metric(name='r2', score=1.0)
>>> metric.score
1.0
>>> metric.name
'r2'
>>> metric.cross_val_scores
array([1., 1.])
>>> metric.std
0.0
```

Method generated by attrs for class Metric.

score_metric(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], x: Union[pandas.core.series.Series, numpy.ndarray], y: Union[pandas.core.series.Series, numpy.ndarray]*) → ml_tooling.metrics.metric.Metric

Calculates the score for this metric. Takes a fitted estimator, x and y values. Scores are calculated with sklearn metrics - using the string defined in *self.metric* to look up the appropriate scoring function.

Parameters

- **estimator** (*Pipeline or BaseEstimator*) – A fitted estimator to score
- **x** (*np.ndarray, pd.DataFrame*) – Features to score model with
- **y** (*np.ndarray, pd.Series*) – Target to score model with

Returns

Return type self

score_metric_cv(*estimator: Union[sklearn.base.BaseEstimator, sklearn.pipeline.Pipeline], x: Union[pandas.core.series.Series, numpy.ndarray], y: Union[pandas.core.series.Series, numpy.ndarray], cv: Any, n_jobs: int = -1, verbose: int = 0*) → ml_tooling.metrics.metric.Metric

Score metric using cross-validation. When scoring with cross_validation, *self.cross_val_scores* is populated with the cross validated scores and *self.score* is set to the mean value of *self.cross_val_scores*. Cross validation can be parallelized by passing the *n_jobs* parameter

Parameters

- **estimator** (*Pipeline or BaseEstimator*) – Fitted estimator to score
- **x** (*np.ndarray or pd.DataFrame*) – Features to use in scoring
- **y** (*np.ndarray or pd.Series*) – Target to use in scoring
- **cv** (*int, BaseCrossValidator*) – If an int is passed, cross-validate using K-Fold with *cv* folds. If BaseCrossValidator is passed, use that object instead
- **n_jobs** (*int*) – Number of jobs to use in parallelizing. Pass None to not do CV in parallel
- **verbose** (*int*) – Verbosity level of output

Returns

Return type self

class ml_tooling.metrics.**Metrics**(*metrics: List[ml_tooling.metrics.metric.Metric]*)

Represents a collection of *Metric*. This is the default object used when scoring an estimator.

There are two alternate constructors: - `from_list()` takes a list of metric names and instantiates one metric per list item - `from_dict()` takes a dictionary of name -> score and instantiates one metric with the given score per dictionary item

Calling either `score_metrics()` or `score_metrics_cv()` will in turn call `score_metric()` or `score_metric_cv()` of each *Metric* in its collection

Examples

To score multiple metrics, create a metrics object from a list and call `score_metrics()` to score all metrics in one operation

We can convert metrics to a dictionary

or a list

Method generated by attrs for class Metrics.

Python Module Index

m

Index

L

M

N

P

R

S

T

V