epftoolbox

Release 1.0

Jul 02, 2021

Contents

This is the documentation of the epftoolbox, the first open-access library for driving research in electricity price forecasting. Its main goal is to make available a set of tools that ensure reproducibility and establish research standards in electricity price forecasting research.

The library contrain three main components:

- The *data management* subpackage, which comprises a module for *processing data* and another module for *dataset extraction*.
- The *models* subpackage, which provides two state of the art forecasting models for electricity price forecasting. The contains a module for the *LEAR* model and another module for the *DNN* model.
- The *evaluation* subpackage, which includes a module for evaluating the performance of the models in terms of *accuracy metrics*, and another module to compare the forecasts of the models via *statistical testing*.

The library is distributed under the AGPL-3.0 License and it is built on top of scikit-learn, tensorflow, keras, hyperopt, statsmodels, numpy, and pandas.

Using the index on the navigation bar or the index below you can navigate through the different library components.

CHAPTER 1

Getting started

1.1 Installation

The library can be easily installed using pip. First clone the library and navigate to the folder:

```
git clone https://github.com/jeslago/epftoolbox.git
cd epftoolbox
```

Then, simply install the library using pip:

```
pip install .
```

1.2 Functionality

The library has three distinct modules: the *data management* module, the *models* module, and the *evaluation* module.

The *first module* provides functionality to manage, process, and obtain data for electricity price forecasting. The module also provides access to data from five different day-ahead electricity markets: EPEX-BE, EPEX-FR, EPEX-DE, NordPool, and PJM markets.

The *second module* grants access to state-of-the-art forecasting methods for day-ahead electricity prices that require no expert knowledge and can be automatically employed. At the moment, the library includes two state-of-the-art models: the *LEAR* model and the *DNN* model.

The *third module* provides with an easy-to-use interface for evaluating forecasts in electricity price forecasting. This module includes both scalar metrics like MAE or MASE as well as statistical tests to evaluate the statistical difference in forecasting performance.

1.3 Using the library

To learn how to use the library, there are three possibilities:

1. Since the library is rather simply, an user can easily read the library documentation of the specific modules that are of interest. The documentation includes explanations of each module and how to use them.
2. As an alternative, there are a number of *examples* that illustrate the most relevant functionalities of the library.
3. Finally, the library and its functionalities are explained in detail in the following article:

- Jesus Lago, Grzegorz Marcjasz, Bart De Schutter, Rafał Weron. "Forecasting day-ahead electricity prices: A review of state-of-the-art algorithms, best practices and an open-access benchmark". *Applied Energy* 2021; 293:116983. [https://doi.org/10.1016/j.apenergy.2021.116983{]}(https://doi.org/10.1016/j.apenergy.2021.116983).

For the most comprehensive introduction to the library, the user should first read the article, then the documentation, and finally go through the available *examples*.

CHAPTER 2

Data management

This subpackage provides an interface to extract data from different day-ahead electricity markets and a module to process the market data before its use in prediction models.

The first functionality is provided by the *data extraction* module, which provides automatic access to the data from five different day-ahead electricity markets as well as an easy-to-use interface to read data from other markets.

The second functionality is provided by the *data wrangling* module, which includes the most common scaling transformations in electricity price forecasting.

2.1 Dataset extraction

This module provides an easy-to-use interface to download data from multiple day-ahead electricity markets using the following database. The module is built around the function `read_data`, and it can be used to obtain the market data from the following periods and day-ahead electricity markets:

Market	Period
Nord pool	01.01.2013 – 24.12.2018
PJM	01.01.2013 – 24.12.2018
EPEX-France	09.01.2011 – 31.12.2016
EPEX-Belgium	09.01.2011 – 31.12.2016
EPEX-Germany	09.01.2012 – 31.12.2017

Besides the data from these five markets, the module also provides an interface to read *csv* files from other markets and transform their data to match the naming requirements of the prediction models in the epftoolbox library. In addition, it also implements an automatic training/testing split based on the testing period under study.

`epftoolbox.data.`**`read_data`**(*path*, *dataset='PJM'*, *years_test=2*, *begin_test_date=None*, *end_test_date=None*)

Function to read and import data from day-ahead electricity markets.

It receives a `dataset` name, and the `path` of the folder where datasets are saved. It reads the file `dataset.csv` in the `path` directory and provides a split between training and testing dataset based on the test dates

provided.

It also names the columns of the training and testing dataset to match the requirements of the prediction models of the library. Namely, assuming that there are *N* exogenous inputs, the columns of the resulting training and testing dataframes are named `['Price', 'Exogenous 1', 'Exogenous 2', ...., 'Exogenous N']`.

If *dataset* is either `"PJM"`, `"NP"`, `"BE"`, `"FR"`, or `"DE"`, the function checks whether `dataset.csv` exists in `path`. If it doesn't exist, it downloads the data from an online database and saves it under the `path` directory. `"PJM"` refes to the Pennsylvania-New Jersey-Maryland market, `"NP"` to the Nord Pool market, and `"BE"`, `"FR"`, and `"DE"` respectively to the EPEX-Belgium, EPEX-France, and EPEX-Germany day-ahead markets.

Note that the data available online for these five markets is limited to certain periods (see the database for further details).

Parameters

- **path** (*str, optional*) – Path where the datasets are stored or, if they do not exist yet, the path where the datasets are to be stored
- **nlayers** (*int, optional*) – Number of hidden layers in the neural network
- **dataset** (*str, optional*) – Name of the dataset/market under study. If it is one one of the standard markets, i.e. `"PJM"`, `"NP"`, `"BE"`, `"FR"`, or `"DE"`, the dataset is automatically downloaded. If the name is different, a dataset with a csv format should be place in the `path`.
- **years_test** (*int, optional*) – Number of years (a year is 364 days) in the test dataset. It is only used if the arguments begin_test_date and end_test_date are not provided.
- **begin_test_date** (*datetime/str, optional*) – Optional parameter to select the test dataset. Used in combination with the argument `end_test_date`. If either of them is not provided, the test dataset is built using the `years_test` argument. `begin_test_date` should either be a string with the following format `"%d/%m/%Y %H:%M"`, or a datetime object.
- **end_test_date** (*datetime/str, optional*) – Optional parameter to select the test dataset. Used in combination with the argument `begin_test_date`. If either of them is not provided, the test dataset is built using the `years_test` argument. `end_test_date` should either be a string with the following format `"%d/%m/%Y %H:%M"`, or a datetime object.

Returns Training dataset, testing dataset

Return type pandas.DataFrame, pandas.DataFrame

Example

```
>>> from epftoolbox.data import read_data
>>> df_train, df_test = read_data(path='.', dataset='PJM', begin_test_date='01-01-
↪2016',
...                               end_test_date='01-02-2016')
Test datasets: 2016-01-01 00:00:00 - 2016-02-01 23:00:00
>>> df_train.tail()
                        Price  Exogenous 1  Exogenous 2
Date
2015-12-31 19:00:00  29.513832     100700.0      13015.0
2015-12-31 20:00:00  28.440134      99832.0      12858.0
2015-12-31 21:00:00  26.701700      97033.0      12626.0
```

(continues on next page)

(continued from previous page)

```
2015-12-31 22:00:00  23.262253      92022.0      12176.0
2015-12-31 23:00:00  22.262431      86295.0      11434.0
>>> df_test.head()
                        Price  Exogenous 1  Exogenous 2
Date
2016-01-01 00:00:00  20.341321      76840.0      10406.0
2016-01-01 01:00:00  19.462741      74819.0      10075.0
2016-01-01 02:00:00  17.172706      73182.0       9795.0
2016-01-01 03:00:00  16.963876      72300.0       9632.0
2016-01-01 04:00:00  17.403722      72535.0       9566.0
>>> df_test.tail()
                        Price  Exogenous 1  Exogenous 2
Date
2016-02-01 19:00:00  28.056729      99400.0      12680.0
2016-02-01 20:00:00  26.916456      97553.0      12495.0
2016-02-01 21:00:00  24.041505      93983.0      12267.0
2016-02-01 22:00:00  22.044896      88535.0      11747.0
2016-02-01 23:00:00  20.593339      82900.0      10974.0
```

2.2 Data wrangling

This module is intended for transforming data into a format that can be read and processed by the prediction models of the epftoolbox library. At the moment, the module is limited to scaling operations.

The module is composed of two components:

- The *DataScaler* class.
- The *scaling* function.

The class *DataScaler* is the main block for performing scaling operations. The class is based on the syntax of the scalers defined in the sklearn.preprocessing module of the scikit-learn library. The class performs some of the standard scaling algorithms in the context of electricity price forecasting:

Besides the class, the module also provides a function *scaling* to scale a list of datasets but estimating the scaler using only one of the datasets. This function is useful when scaling the training, validation, and test dataset. In this scenario, to have a realistic evaluation, one would ideally estimate the scaler using the training dataset and simply transform the other two.

---

class epftoolbox.data.**DataScaler**(*normalize*)

Class to perform data scaling operations

The scaling technique is defined by the `normalize` parameter which takes one of the following values:

- `'Norm'` for normalizing the data to the interval [0, 1].
- `'Norm1'` for normalizing the data to the interval [-1, 1].
- `'Std'` for standarizing the data to follow a normal distribution.
- `'Median'` for normalizing the data based on the median as defined in as defined in here.
- `'Invariant'` for scaling the data based on the asinh transformation (a variance stabilizing transformations) as defined in here.

This class follows the same syntax of the scalers defined in the sklearn.preprocessing module of the scikit-learn library

Parameters **normalize** (*str*) – Type of scaling to be performed. Possible values are 'Norm', 'Norm1', 'Std', 'Median', or 'Invariant'

Example

```
>>> from epftoolbox.data import read_data
>>> from epftoolbox.data import DataScaler
>>> df_train, df_test = read_data(path='.', dataset='PJM', begin_test_date='01-01-
↪2016', end_test_date='01-02-2016')
Test datasets: 2016-01-01 00:00:00 - 2016-02-01 23:00:00
>>> df_train.tail()
                        Price  Exogenous 1  Exogenous 2
Date
2015-12-31 19:00:00  29.513832     100700.0      13015.0
2015-12-31 20:00:00  28.440134      99832.0      12858.0
2015-12-31 21:00:00  26.701700      97033.0      12626.0
2015-12-31 22:00:00  23.262253      92022.0      12176.0
2015-12-31 23:00:00  22.262431      86295.0      11434.0
>>> df_test.head()
                        Price  Exogenous 1  Exogenous 2
Date
2016-01-01 00:00:00  20.341321      76840.0      10406.0
2016-01-01 01:00:00  19.462741      74819.0      10075.0
2016-01-01 02:00:00  17.172706      73182.0       9795.0
2016-01-01 03:00:00  16.963876      72300.0       9632.0
2016-01-01 04:00:00  17.403722      72535.0       9566.0
>>> Xtrain = df_train.values
>>> Xtest = df_train.values
>>> scaler = DataScaler('Norm')
>>> Xtrain_scaled = scaler.fit_transform(Xtrain)
>>> Xtest_scaled = scaler.transform(Xtest)
>>> Xtrain_inverse = scaler.inverse_transform(Xtrain_scaled)
>>> Xtest_inverse = scaler.inverse_transform(Xtest_scaled)
>>> Xtrain[:3,:]
array([[2.5464211e+01, 8.5049000e+04, 1.1509000e+04],
       [2.3554578e+01, 8.2128000e+04, 1.0942000e+04],
       [2.2122277e+01, 8.0729000e+04, 1.0639000e+04]])
>>> Xtrain_scaled[:3,:]
array([[0.03833877, 0.2736787 , 0.28415155],
       [0.03608228, 0.24425597, 0.24633138],
       [0.03438982, 0.23016409, 0.2261206 ]])
>>> Xtrain_inverse[:3,:]
array([[2.5464211e+01, 8.5049000e+04, 1.1509000e+04],
       [2.3554578e+01, 8.2128000e+04, 1.0942000e+04],
       [2.2122277e+01, 8.0729000e+04, 1.0639000e+04]])
>>> Xtest[:3,:]
array([[2.5464211e+01, 8.5049000e+04, 1.1509000e+04],
       [2.3554578e+01, 8.2128000e+04, 1.0942000e+04],
       [2.2122277e+01, 8.0729000e+04, 1.0639000e+04]])
>>> Xtest_scaled[:3,:]
array([[0.03833877, 0.2736787 , 0.28415155],
       [0.03608228, 0.24425597, 0.24633138],
       [0.03438982, 0.23016409, 0.2261206 ]])
>>> Xtest_inverse[:3,:]
array([[2.5464211e+01, 8.5049000e+04, 1.1509000e+04],
       [2.3554578e+01, 8.2128000e+04, 1.0942000e+04],
       [2.2122277e+01, 8.0729000e+04, 1.0639000e+04]])
```

Methods

fit_transform(dataset)	Method that estimates an scaler object using the data in `dataset` and scales the data in `dataset`
inverse_transform(dataset)	Method that inverse-scale the data in `dataset`
transform(dataset)	Method that scales the data in `dataset`

fit_transform(*dataset*)
Method that estimates an scaler object using the data in `dataset` and scales the data in `dataset`

Parameters **dataset** (*numpy.array*) – Dataset used to estimate the scaler

Returns Scaled data

Return type numpy.array

inverse_transform(*dataset*)
Method that inverse-scale the data in `dataset`

It must be called after calling the *fit_transform* method for estimating the scaler

Parameters **dataset** (*numpy.array*) – Dataset to be scaled

Returns Inverse-scaled data

Return type numpy.array

transform(*dataset*)
Method that scales the data in `dataset`

It must be called after calling the *fit_transform* method for estimating the scaler :param dataset: Dataset to be scaled :type dataset: numpy.array

Returns Scaled data

Return type numpy.array

---

epftoolbox.data.**scaling**(*datasets*, *normalize*)
Function that scales data and returns the scaled data and the *DataScaler* used for scaling.

It rescales all the datasets contained in the list `datasets` using the first dataset as reference. For example, if `datasets=[X_1, X_2, X_3]`, the function estimates a *DataScaler* object using the array `X_1`, and transform `X_1`, `X_2`, and `X_3` using the *DataScaler* object.

Each dataset must be a numpy.array and it should have the same column-dimensions. For example, if `datasets=[X_1, X_2, X_3]`, `X_1` must be a numpy.array of size `[n_1, m]`, `X_2` of size `[n_2, m]`, and `X_3` of size `[n_3, m]`, where `n_1`, `n_2`, `n_3` can be different.

The scaling technique is defined by the `normalize` parameter which takes one of the following values:

- `'Norm'` for normalizing the data to the interval [0, 1].
- `'Norm1'` for normalizing the data to the interval [-1, 1].
- `'Std'` for standarizing the data to follow a normal distribution.
- `'Median'` for normalizing the data based on the median as defined in as defined in here.
- `'Invariant'` for scaling the data based on the asinh transformation (a variance stabilizing transformations) as defined in here.

The function returns the scaled data together with a *DataScaler* object representing the scaling. This object can be used to scale other dataset using the same rules or to inverse-transform the data.

Parameters

- **datasets** (*list*) – List of numpy.array objects to be scaled.
- **normalize** (*str*) – Type of scaling to be performed. Possible values are `'Norm'`, `'Norm1'`, `'Std'`, `'Median'`, or `'Invariant'`

Returns List of scaled datasets and the *DataScaler* object used for scaling. Each dataset in the list is a numpy.array.

Return type List, *DataScaler*

Example

```
>>> from epftoolbox.data import read_data
>>> from epftoolbox.data import scaling
>>> df_train, df_test = read_data(path='.', dataset='PJM', begin_test_date='01-01-
↪2016', end_test_date='01-02-2016')
Test datasets: 2016-01-01 00:00:00 - 2016-02-01 23:00:00
>>> df_train.tail()
                        Price  Exogenous 1  Exogenous 2
Date
2015-12-31 19:00:00  29.513832     100700.0      13015.0
2015-12-31 20:00:00  28.440134      99832.0      12858.0
2015-12-31 21:00:00  26.701700      97033.0      12626.0
2015-12-31 22:00:00  23.262253      92022.0      12176.0
2015-12-31 23:00:00  22.262431      86295.0      11434.0
>>> df_test.head()
                        Price  Exogenous 1  Exogenous 2
Date
2016-01-01 00:00:00  20.341321      76840.0      10406.0
2016-01-01 01:00:00  19.462741      74819.0      10075.0
2016-01-01 02:00:00  17.172706      73182.0       9795.0
2016-01-01 03:00:00  16.963876      72300.0       9632.0
2016-01-01 04:00:00  17.403722      72535.0       9566.0
>>> Xtrain = df_train.values
>>> Xtest = df_train.values
>>> [Xtrain_scaled, Xtest_scaled], scaler = scaling([Xtrain,Xtest],'Norm')
>>> Xtrain[:3,:]
array([[2.5464211e+01, 8.5049000e+04, 1.1509000e+04],
       [2.3554578e+01, 8.2128000e+04, 1.0942000e+04],
       [2.2122777e+01, 8.0729000e+04, 1.0639000e+04]])
>>> Xtrain_scaled[:3,:]
array([[0.03833877, 0.2736787 , 0.28415155],
       [0.03608228, 0.24425597, 0.24633138],
       [0.03438982, 0.23016409, 0.2261206 ]])
>>> Xtest[:3,:]
array([[2.5464211e+01, 8.5049000e+04, 1.1509000e+04],
       [2.3554578e+01, 8.2128000e+04, 1.0942000e+04],
       [2.2122777e+01, 8.0729000e+04, 1.0639000e+04]])
>>> Xtest_scaled[:3,:]
array([[0.03833877, 0.2736787 , 0.28415155],
       [0.03608228, 0.24425597, 0.24633138],
       [0.03438982, 0.23016409, 0.2261206 ]])
```

(continues on next page)

(continued from previous page)

```
>>> type(scaler)
<class 'epftoolbox.data._wrangling.DataScaler'>
```

CHAPTER 3

Forecasting models

This subpackage provides an easy interface to two state-of-the-art forecasting models in the field of electricity price forecasting:

3.1 LEAR

The LEAR model is a parameter-rich ARX model estimated using the LASSO as an implicit feature selection that was originally proposed by Uniejewski (2016). It has been used in multiple studies and it has often shown state-of-the-art results in electricity price forecasting, e.g. see Uniejewski (2016) or Lago (2018).

The LEAR model is provided in the library as a sigle *LEAR* class. The class receives as parameter the calibration window of the method, and has three four main function: a function to recalibrate the model, a function to make predictions, a function to recalibrate and predict, and a function that can perform daily recalibration and prediction using pandas DataFrames.

Besides the LEAR class, the library also includes the *evaluate_lear_in_test_dataset* function. This function can be used as a simplified interface to evaluate a pandas DataFrame by simply specific the dates of the training and test datasets.

The library also includes a couple of *LEAR Examples* to get users familiar with the syntax and capabilities of the model.

3.1.1 epftoolbox.models.LEAR

class epftoolbox.models.**LEAR**(*calibration_window=1092*)

Class to build a LEAR model, recalibrate it, and use it to predict DA electricity prices.

An example on how to use this class is provided *here*.

Parameters **calibration_window** (*int, optional*) – Calibration window (in days) for the LEAR model.

Methods

predict(X)	Function that makes a prediction using some given inputs.
recalibrate(Xtrain, Ytrain)	Function to recalibrate the LEAR model.
recalibrate_and_forecast_next_day(df, ...)	Easy-to-use interface for daily recalibration and forecasting of the LEAR model.
recalibrate_predict(Xtrain, Ytrain, Xtest)	Function that first recalibrates the LEAR model and then makes a prediction.

predict(*X*)

Function that makes a prediction using some given inputs.

Parameters **X** (*numpy.array*) – Input of the model.

Returns An array containing the predictions.

Return type numpy.array

recalibrate(*Xtrain*, *Ytrain*)

Function to recalibrate the LEAR model.

It uses a training (Xtrain, Ytrain) pair for recalibration

Parameters

- **Xtrain** (*numpy.array*) – Input in training dataset. It should be of size *[n,m]* where *n* is the number of days in the training dataset and *m* the number of input features
- **Ytrain** (*numpy.array*) – Output in training dataset. It should be of size *[n,24]* where *n* is the number of days in the training dataset and 24 are the 24 prices of each day

Returns The prediction of day-ahead prices after recalibrating the model

Return type numpy.array

recalibrate_and_forecast_next_day(*df*, *calibration_window*, *next_day_date*)

Easy-to-use interface for daily recalibration and forecasting of the LEAR model.

The function receives a pandas dataframe and a date. Usually, the data should correspond with the date of the next-day when using for daily recalibration.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe of historical data containing prices and *N* exogenous inputs. The index of the dataframe should be dates with hourly frequency. The columns should have the following names `['Price', 'Exogenous 1', 'Exogenous 2', ...., 'Exogenous N']`.
- **calibration_window** (*int*) – Calibration window (in days) for the LEAR model.
- **next_day_date** (*datetime*) – Date of the day-ahead.

Returns The prediction of day-ahead prices.

Return type numpy.array

recalibrate_predict(*Xtrain*, *Ytrain*, *Xtest*)

Function that first recalibrates the LEAR model and then makes a prediction.

The function receives the training dataset, and trains the LEAR model. Then, using the inputs of the test dataset, it makes a new prediction.

Parameters

- **Xtrain** (*numpy.array*) – Input of the training dataset.
- **Xtest** (*numpy.array*) – Input of the test dataset.
- **Ytrain** (*numpy.array*) – Output of the training dataset.

Returns An array containing the predictions in the test dataset.

Return type numpy.array

3.1.2 epftoolbox.models.evaluate_lear_in_test_dataset

epftoolbox.models.**evaluate_lear_in_test_dataset**(*path_datasets_folder='./datasets'*, *path_recalibration_folder='./experimental_files'*, *dataset='PJM'*, *years_test=2*, *calibration_window=1092*, *begin_test_date=None*, *end_test_date=None*)

Function for easy evaluation of the LEAR model in a test dataset using daily recalibration.

The test dataset is defined by a market name and the test dates dates. The function generates the test and training datasets, and evaluates a LEAR model considering daily recalibration.

An example on how to use this function is provided *here*.

Parameters

- **path_datasets_folder** (*str, optional*) – path where the datasets are stored or, if they do not exist yet, the path where the datasets are to be stored.
- **path_recalibration_folder** (*str, optional*) – path to save the files of the experiment dataset.
- **dataset** (*str, optional*) – Name of the dataset/market under study. If it is one one of the standard markets, i.e. `"PJM"`, `"NP"`, `"BE"`, `"FR"`, or `"DE"`, the dataset is automatically downloaded. If the name is different, a dataset with a csv format should be place in the `path_datasets_folder`.
- **years_test** (*int, optional*) – Number of years (a year is 364 days) in the test dataset. It is only used if the arguments `begin_test_date` and `end_test_date` are not provided.
- **calibration_window** (*int, optional*) – Number of days used in the training dataset for recalibration.
- **begin_test_date** (*datetime/str, optional*) – Optional parameter to select the test dataset. Used in combination with the argument `end_test_date`. If either of them is not provided, the test dataset is built using the `years_test` argument. `begin_test_date` should either be a string with the following format `"%d/%m/%Y %H:%M"`, or a datetime object.
- **end_test_date** (*datetime/str, optional*) – Optional parameter to select the test dataset. Used in combination with the argument `begin_test_date`. If either of them is not provided, the test dataset is built using the `years_test` argument. `end_test_date` should either be a string with the following format `"%d/%m/%Y %H:%M"`, or a datetime object.

Returns A dataframe with all the predictions in the test dataset. The dataframe is also written to path_recalibration_folder.

Return type pandas.DataFrame

3.2 DNN

The DNN model is a deep neural network tailored to electricity price forecasting whose input features and hyperparameters are optimized for each market without the need of expert knowledge. The DNN model was originally proposed by Lago (2018) in a study where it was shown to obtain state-of-the-art results. Although more complex, it is often more accurate than individual *LEAR* models.

The module is built around the *DNNModel* class. This class represents a basic DNN model based on keras and tensorflow. While the model can be used standalone to train and predict a DNN, it is intended to be used within the *hyperparameter_optimizer* function and the *DNN* class. These two elements represent the main two functionalities of the module.

In particular, the *hyperparameter_optimizer* function provides an interface to optimize the optimal hyperparameter and features of the DNN model. Then, the *DNN* class considers the output of the *hyperparameter_optimizer* function, i.e. the set of optimal hyperparameters and features, and provides an interface to perform recalibration and new predictions. The class extends the functionality of the `DNNModel` class by providing an interface to extract the best set of hyperparameters, and to perform recalibration before every prediction.

The module also includes the *evaluate_dnn_in_test_dataset* function. This function can be used as a simplified interface to evaluate a test period in a dataset that is built using a pandas DataFrame.

The library also includes several *DNN Examples* to get users familiar with the syntax and capabilities of the model.

3.2.1 epftoolbox.models.DNNModel

class epftoolbox.models.**DNNModel**(*neurons*, *n_features*, *outputShape=24*, *dropout=0*, *batch_normalization=False*, *lr=None*, *verbose=False*, *epochs_early_stopping=40*, *scaler=None*, *loss='mae'*, *optimizer='adam'*, *activation='relu'*, *initializer='glorot_uniform'*, *regularization=None*, *lambda_reg=0*)

Basic DNN model based on keras and tensorflow.

The model can be used standalone to train and predict a DNN using its fit/predict methods. However, it is intended to be used within the *hyperparameter_optimizer* method and the *DNN* class. The former obtains a set of best hyperparameter using the *DNNModel* class. The latter employes the set of best hyperparameters to recalibrate a *DNNModel* object and make predictions.

Parameters

- **neurons** (*list*) – List containing the number of neurons in each hidden layer. E.g. if `len(neurons)` is 2, the DNN model has an input layer of size `n_features`, two hidden layers, and an output layer of size `outputShape`.
- **n_features** (*int*) – Number of input features in the model. This number defines the size of the input layer.
- **outputShape** (*int, optional*) – Default number of output neurons. It is 24 as it is the default in most day-ahead markets.
- **dropout** (*float, optional*) – Number between [0, 1] that selects the percentage of dropout. A value of 0 indicates no dropout.
- **batch_normalization** (*bool, optional*) – Boolean that selects whether batch normalization is considered.

- **lr** (*float, optional*) – Learning rate for optimizer algorithm. If none provided, the default one is employed (see the keras documentation for the default learning rates of each algorithm).
- **verbose** (*bool, optional*) – Boolean that controls the logs. If set to true, a minimum amount of information is displayed.
- **epochs_early_stopping** (*int, optional*) – Number of epochs used in early stopping to stop training. When no improvement is observed in the validation dataset after `epochs_early_stopping` epochs, the training stops.
- **scaler** (*epftoolbox.data.DataScaler*, optional) – Scaler object to invert-scale the output of the neural network if the neural network is trained with scaled outputs.
- **loss** (*str, optional*) – Loss to be used when training the neural network. Any of the regression losses defined in keras can be used.
- **optimizer** (*str, optional*) – Name of the optimizer when training the DNN. See the keras documentation for a list of optimizers.
- **activation** (*str, optional*) – Name of the activation function in the hidden layers. See the keras documentation for a list of activation function.
- **initializer** (*str, optional*) – Name of the initializer function for the weights of the neural network. See the keras documentation for a list of initializer functions.
- **regularization** (*None, optional*) – Name of the regularization technique. It can can have three values `'l2'` for l2-norm regularization, `'l1'` for l1-norm regularization, or `None` for no regularization .
- **lambda_reg** (*int, optional*) – The weight for regulization if `regularization` is `'l2'` or `'l1'`.

Methods

clear_session()	Method to clear the tensorflow session.
fit(trainX, trainY, valX, valY)	Method to estimate the DNN model.
predict(X)	Method to make a prediction after the DNN is trained.

clear_session()
Method to clear the tensorflow session.

It is used in the *DNN* class during recalibration to avoid RAM memory leakages. In particular, if the DNN is retrained continuosly, at each step tensorflow slightly increases the total RAM usage.

fit(*trainX*, *trainY*, *valX*, *valY*)
Method to estimate the DNN model.

Parameters

- **trainX** (*numpy.array*) – Inputs fo the training dataset.
- **trainY** (*numpy.array*) – Outputs fo the training dataset.
- **valX** (*numpy.array*) – Inputs fo the validation dataset used for early-stopping.
- **valY** (*numpy.array*) – Outputs fo the validation dataset used for early-stopping.

predict(*X*)
Method to make a prediction after the DNN is trained.

Parameters **X** (*numpy.array*) – Input to the DNN. It has to be of size *[n, n_features]* where *n* can be any integer, and *n_features* is the attribute of the DNN representing the number of input features.

Returns Output of the DNN after making the prediction.

Return type numpy.array

3.2.2 epftoolbox.models.hyperparameter_optimizer

epftoolbox.models.**hyperparameter_optimizer**(*path_datasets_folder='./datasets'*, *path_hyperparameters_folder='./experimental_files'*, *new_hyperopt=1*, *max_evals=1500*, *nlayers=2*, *dataset='PJM'*, *years_test=2*, *calibration_window=4*, *shuffle_train=1*, *data_augmentation=0*, *experiment_id=None*, *begin_test_date=None*, *end_test_date=None*)

Function to optimize the hyperparameters and input features of the DNN. An example on how to use this function is provided *here*.

Parameters

- **path_datasets_folder** (*str, optional*) – Path to read and store datasets.
- **path_hyperparameters_folder** (*str, optional*) – Path to read and store trials files from hyperopt.
- **new_hyperopt** (*bool, optional*) – Boolean that decides whether to start a new hyperparameter optimization or re-start an existing one.
- **max_evals** (*int, optional*) – Maximum number of iterations for hyperopt.
- **nlayers** (*int, optional*) – Number of layers of the DNN model.
- **dataset** (*str, optional*) – Name of the dataset/market under study. If it is one one of the standard markets, i.e. `"PJM"`, `"NP"`, `"BE"`, `"FR"`, or `"DE"`, the dataset is automatically downloaded. If the name is different, a dataset with a csv format should be place in the `path_datasets_folder`.
- **years_test** (*int, optional*) – Number of years (a year is 364 days) in the test dataset. It is only used if the arguments `begin_test_date` and `end_test_date` are not provided.
- **calibration_window** (*int, optional*) – Calibration window used for training the models.
- **shuffle_train** (*bool, optional*) – Boolean that selects whether the validation and training datasets are shuffled. Based on empirical results, this configuration does not play a role when selecting the hyperparameters and features. However, it is important when recalibrating the DNN model.
- **data_augmentation** (*bool, optional*) – Boolean that selects whether a data augmentation technique for DNNs is used. Based on empirical results, for some markets data augmentation might improve forecasting accuracy at the expense of higher computational costs.
- **experiment_id** (*None, optional*) – Unique identifier to save/read the trials file. If not provided, the current date is used as identifier.

- **begin_test_date** (*datetime/str, optional*) – Optional parameter to select the test dataset. Used in combination with the argument `end_test_date`. If either of them is not provided, the test dataset is built using the `years_test` argument. `begin_test_date` should either be a string with the following format `"%d/%m/%Y %H:%M"`, or a datetime object.
- **end_test_date** (*datetime/str, optional*) – Optional parameter to select the test dataset. Used in combination with the argument `begin_test_date`. If either of them is not provided, the test dataset is built using the `years_test` argument. `end_test_date` should either be a string with the following format `"%d/%m/%Y %H:%M"`, or a datetime object.

3.2.3 epftoolbox.models.DNN

class epftoolbox.models.**DNN**(*experiment_id*, *path_hyperparameter_folder='./experimental_files'*, *nlayers=2*, *dataset='PJM'*, *years_test=2*, *shuffle_train=1*, *data_augmentation=0*, *calibration_window=4*)

DNN for electricity price forecasting.

It considers a set of best hyperparameters, it recalibrates a *DNNModel* based on these hyperparameters, and makes new predictions.

The difference w.r.t. the *DNNModel* class lies on the functionality. The *DNNModel* class provides a simple interface to build a keras DNN model which is limited to fit and predict methods. This class extends the functionality by providing an interface to extract the best set of hyperparameters, and to perform recalibration before every prediction.

Note that before using this class, a hyperparameter optimization run must be done using the *hyperparameter_optimizer* function. Such hyperparameter optimization must be done using the same parameters: `nlayers`, `dataset`, `years_test`, `shuffle_train`, `data_augmentation`, and `calibration_window`

An example on how to use this class is provided *here*.

Parameters

- **experiment_id** (*str*) – Unique identifier to read the trials file. In particular, every hyperparameter optimization set has an unique identifier associated with. See *hyperparameter_optimizer* for further details
- **path_hyperparameter_folder** (*str, optional*) – Path of the folder containing the trials file with the optimal hyperparameters
- **nlayers** (*int, optional*) – Number of layers of the DNN model
- **dataset** (*str, optional*) – Name of the dataset/market under study. If it is one one of the standard markets, i.e. `"PJM"`, `"NP"`, `"BE"`, `"FR"`, or `"DE"`, the dataset is automatically downloaded. If the name is different, a dataset with a csv format should be place in the `path_datasets_folder`.
- **years_test** (*int, optional*) – Number of years (a year is 364 days) in the test dataset. This is necesary to extract the correct hyperparameter trials file
- **shuffle_train** (*bool, optional*) – Boolean that selects whether the validation and training datasets were shuffled when performing the hyperparameter optimization. Note that it does not select whether shuffling is used for recalibration as for recalibration the validation and the training datasets are always shuffled.
- **data_augmentation** (*bool, optional*) – Boolean that selects whether a data augmentation technique for electricity price forecasting is employed

- **calibration_window** (*int, optional*) – Number of days used in the training/validation dataset for recalibration

Methods

predict(X)	Method that makes a prediction using some given inputs
recalibrate(Xtrain, Ytrain, Xval, Yval)	Method that recalibrates the model.
recalibrate_and_forecast_next_day(df, ...)	Method that builds an easy-to-use interface for daily recalibration and forecasting of the DNN model
recalibrate_predict(Xtrain, Ytrain, Xval, ...)	Method that first recalibrates the DNN model and then makes a prediction.

predict(*X*)

Method that makes a prediction using some given inputs

Parameters **X** (*numpy.array*) – Input of the model

Returns An array containing the predictions

Return type numpy.array

recalibrate(*Xtrain*, *Ytrain*, *Xval*, *Yval*)

Method that recalibrates the model.

The method receives the training and validation dataset, and trains a *DNNModel* model using the set of optimal hyperparameters that are found in `path_hyperparameter_folder` and that are defined by the class attributes: `experiment_id`, `nlayers`, `dataset`, `years_test`, `shuffle_train`, `data_augmentation`, and `calibration_window`

Parameters

- **Xtrain** (*numpy.array*) – Input of the training dataset
- **Xval** (*numpy.array*) – Input of the validation dataset
- **Ytrain** (*numpy.array*) – Output of the training dataset
- **Yval** (*numpy.array*) – Output of the validation dataset

recalibrate_and_forecast_next_day(*df*, *next_day_date*)

Method that builds an easy-to-use interface for daily recalibration and forecasting of the DNN model

The method receives a pandas dataframe `df` and a day `next_day_date`. Then, it recalibrates the model using data up to the day before `next_day_date` and makes a prediction for day `next_day_date`.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe of historical data containing prices and N exogenous inputs. The index of the dataframe should be dates with hourly frequency. The columns should have the following names ['Price', 'Exogenous 1', 'Exogenous 2', ..., 'Exogenous N']
- **next_day_date** (*TYPE*) – Date of the day-ahead

Returns An array containing the predictions in the provided date

Return type numpy.array

recalibrate_predict(*Xtrain*, *Ytrain*, *Xval*, *Yval*, *Xtest*)

Method that first recalibrates the DNN model and then makes a prediction.

The method receives the training and validation dataset, and trains a *DNNModel* model using the set of optimal hyperparameters. Then, using the inputs of the test dataset, it makes a new prediction.

Parameters

- **Xtrain** (*numpy.array*) – Input of the training dataset
- **Xval** (*numpy.array*) – Input of the validation dataset
- **Xtest** (*numpy.array*) – Input of the test dataset
- **Ytrain** (*numpy.array*) – Output of the training dataset
- **Yval** (*numpy.array*) – Output of the validation dataset

Returns An array containing the predictions in the test dataset

Return type numpy.array

3.2.4 epftoolbox.models.evaluate_dnn_in_test_dataset

epftoolbox.models.**evaluate_dnn_in_test_dataset**(*experiment_id*, *path_datasets_folder='./datasets'*, *path_hyperparameter_folder='./experimental_files'*, *path_recalibration_folder='./experimental_files'*, *nlayers=2*, *dataset='PJM'*, *years_test=2*, *shuffle_train=True*, *data_augmentation=0*, *calibration_window=4*, *new_recalibration=False*, *begin_test_date=None*, *end_test_date=None*)

Function for easy evaluation of the DNN model in a test dataset using daily recalibration.

The test dataset is defined by a market name and the test dates dates. The function generates the test and training datasets, and evaluates a DNN model considering daily recalibration and an optimal set of hyperparameters.

Note that before using this class, a hyperparameter optimization run must be done using the *hyperparameter_optimizer* function. Moreover, the hyperparameter optimization must be done using the same parameters: `nlayers`, `dataset`, `shuffle_train`, `data_augmentation`, `calibration_window`, and either the `years_test` or the same `begin_test_date`/`end_test_date`

An example on how to use this function is provided *here*.

Parameters

- **experiment_id** (*str*) – Unique identifier to read the trials file. In particular, every hyperparameter optimization set has an unique identifier associated with. See *hyperparameter_optimizer* for further details
- **path_datasets_folder** (*str, optional*) – Path where the datasets are stored or, if they do not exist yet, the path where the datasets are to be stored
- **path_hyperparameter_folder** (*str, optional*) – Path of the folder containing the trials file with the optimal hyperparameters
- **path_recalibration_folder** (*str, optional*) – Path to save the forecast of the test dataset
- **nlayers** (*int, optional*) – Number of hidden layers in the neural network

- **dataset** (*str, optional*) – Name of the dataset/market under study. If it is one one of the standard markets, i.e. `"PJM"`, `"NP"`, `"BE"`, `"FR"`, or `"DE"`, the dataset is automatically downloaded. If the name is different, a dataset with a csv format should be place in the `path_datasets_folder`.
- **years_test** (*int, optional*) – Number of years (a year is 364 days) in the test dataset. It is only used if the arguments begin_test_date and end_test_date are not provided.
- **begin_test_date** (*datetime/str, optional*) – Optional parameter to select the test dataset. Used in combination with the argument end_test_date. If either of them is not provided, the test dataset is built using the years_test argument. begin_test_date should either be a string with the following format d/m/Y H:M, or a datetime object
- **end_test_date** (*datetime/str, optional*) – Optional parameter to select the test dataset. Used in combination with the argument begin_test_date. If either of them is not provided, the test dataset is built using the years_test argument. end_test_date should either be a string with the following format d/m/Y H:M, or a datetime object
- **shuffle_train** (*bool, optional*) – Boolean that selects whether the validation and training datasets were shuffled when performing the hyperparameter optimization. Note that it does not select whether shuffling is used for recalibration as for recalibration the validation and the training datasets are always shuffled.
- **data_augmentation** (*bool, optional*) – Boolean that selects whether a data augmentation technique for electricity price forecasting is employed
- **calibration_window** (*int, optional*) – Number of days used in the training/validation dataset for recalibration
- **new_recalibration** (*bool, optional*) – Boolean that selects whether a new recalibration is performed or the function re-starts an old one. To restart an old one, the .csv file with the forecast must exist in the `path_recalibration_folder` folder

Returns A dataframe with all the predictions in the test dataset. The dataframe is also written to the folder `path_recalibration_folder`

Return type pandas.DataFrame

For the *LEAR* model, the subpackage provides an interface to perform estimation, daily recalibration, and prediction. For *DNN* model, it provides an interface to perform estimation, hyperparameter optimization, daily recalibration, and prediction.

A more detailed explanations of the models can be obtained in:

- Jesus Lago, Grzegorz Marcjasz, Bart De Schutter, Rafał Weron. "Forecasting day-ahead electricity prices: A review of state-of-the-art algorithms, best practices and an open-access benchmark". *Applied Energy* 2021; 293:116983. [https://doi.org/10.1016/j.apenergy.2021.116983{]}(https://doi.org/10.1016/j.apenergy.2021.116983).

CHAPTER 4

Model evaluation

This subpackage provides a set of tools to evaluate forecasts and forecasting models in terms of accuracy metrics and statistical tests.

The first subset of tools, which is provided by the *accuracy metrics* module, evaluate the errors of the predictions based on a single value. They analyze how far the predictions are from the mean or median of the real prices. Examples of metrics are the mean absolute percentage error (MAPE) or the relative mean absolute error (rMAE).

The second subset of tools, which is provided by the *statistical test* module, allows comparison between models by analyzing whether the difference in accuracy in the forecasts of the models is statistically significant. Unlike accuracy metrics, statistical tests allow to infer whether the difference in accuracy does really exist and it is not simply due to random differences between the forecasts.

4.1 Accuracy Metrics

This module provides an easy-to-use interface to the most common and most suitable accuracy metrics in the context of day-ahead prices:

4.1.1 MAE

In the field of electricity price forecasting, one of the most widely used metrics to measure the accuracy of point forecasts is the mean absolute error (MAE):

$$\mathrm{MAE} = \frac{1}{N}\sum_{k=1}^{N} |p_k - \hat{p}_k|, \tag{4.1}$$

(4.2)

This metric computes the average absolute error between the predicted prices and the real prices. Predictive models that minimize the MAE lead to predictions of the median of the prices. Despite its popularity, the *MAE* is not always very informative as absolute errors are hard to compare between different datasets.

epftoolbox.evaluation.MAE

epftoolbox.evaluation.**MAE**(*p_real*, *p_pred*)

Function that computes the mean absolute error (MAE) between two forecasts:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}\left|p_{\text{real}}[i] - p_{\text{pred}}[i]\right|$$

`p_real` and `p_pred` can either be of shape $(n_{\text{days}}, n_{\text{prices/day}})$, $(n_{\text{prices}}, 1)$, or $(n_{\text{prices}},)$ where $n_{\text{prices}} = n_{\text{days}} \cdot n_{\text{prices/day}}$.

Parameters

- **p_real** (*numpy.ndarray, pandas.DataFrame, pandas.Series*) – Array/dataframe containing the real prices.
- **p_pred** (*numpy.ndarray, pandas.DataFrame, pandas.Series*) – Array/dataframe containing the predicted prices.

Returns The mean absolute error (MAE).

Return type float

Example

```
>>> from epftoolbox.evaluation import MAE
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
>>> forecast = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
...                         'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
```

```
>>>
>>> # Transforming indices to datetime format
>>> forecast.index = pd.to_datetime(forecast.index)
>>>
>>> # Reading data from the NP market
>>> _, df_test = read_data(path='.', dataset='NP', begin_test_date=forecast.
↪index[0],
...                         end_test_date=forecast.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
>>>
>>> # Extracting forecast of DNN ensemble and display
>>> fc_DNN_ensemble = forecast.loc[:, ['DNN Ensemble']]
>>>
>>> # Extracting real price and display
>>> real_price = df_test.loc[:, ['Price']]
>>>
>>> # Building the same datasets with shape (ndays, n_prices/day)
>>> # instead of shape (nprices, 1) and display
>>> fc_DNN_ensemble_2D = pd.DataFrame(fc_DNN_ensemble.values.reshape(-1, 24),
...                                   index=fc_DNN_ensemble.index[::24],
```

(continues on next page)

(continued from previous page)

```
...                                     columns=['h' + str(hour) for hour in␣
↪range(24)])
>>> real_price_2D = pd.DataFrame(real_price.values.reshape(-1, 24),
...                              index=real_price.index[::24],
...                              columns=['h' + str(hour) for hour in range(24)])
>>> fc_DNN_ensemble_2D.head()
                  h0         h1         h2  ...        h21        h22        h23
2016-12-27  24.349676  23.127774  22.208617  ...  27.686771  27.045763  25.724071
2016-12-28  25.453866  24.707317  24.452384  ...  29.424558  28.627130  27.321902
2016-12-29  28.209516  27.715400  27.182692  ...  28.473288  27.926241  27.153401
2016-12-30  28.002935  27.467572  27.028558  ...  29.086532  28.518688  27.738548
2016-12-31  25.732282  24.668331  23.951569  ...  26.965008  26.450995  25.637346
```

According to the paper, the MAE of the DNN ensemble for the NP market is 1.667 Let's test the metric for different conditions

```
>>> # Evaluating MAE when real price and forecasts are both dataframes
>>> MAE(p_pred=fc_DNN_ensemble, p_real=real_price)
1.6670355192007669
>>>
>>> # Evaluating MAE when real price and forecasts are both numpy arrays
>>> MAE(p_pred=fc_DNN_ensemble.values, p_real=real_price.values)
1.6670355192007669
>>>
>>> # Evaluating MAE when input values are of shape (ndays, n_prices/day)
>>> # instead of shape (nprices, 1)
>>> # Dataframes
>>> MAE(p_pred=fc_DNN_ensemble_2D, p_real=real_price_2D)
1.6670355192007669
>>> # Numpy arrays
>>> MAE(p_pred=fc_DNN_ensemble_2D.values, p_real=real_price_2D.values)
1.6670355192007669
>>>
>>> # Evaluating MAE when input values are of shape (nprices,)
>>> # instead of shape (nprices, 1)
>>> # Pandas Series
>>> MAE(p_pred=fc_DNN_ensemble.loc[:, 'DNN Ensemble'],
...     p_real=real_price.loc[:, 'Price'])
1.6670355192007669
>>> # Numpy arrays
>>> MAE(p_pred=fc_DNN_ensemble.values.squeeze(),
...     p_real=real_price.values.squeeze())
1.6670355192007669
```

4.1.2 RMSE

In the field of electricity price forecasting, one of the most widely used metrics to measure the accuracy of point forecasts is the root mean square erro (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(p_k - \hat{p}_k)^2}, \tag{4.3}$$

(4.4)

This metric computes the square root of the average of the square errors between the predicted prices and the real prices. Predictive models that minimize the RMSE lead to predictions of the mean of the prices.

Despite its popularity,RMSE is not always very informative as absolute errors are hard to compare between different datasets. In addition, it has the disadvantage of not representing accurately the underlying problem of electricity price forecasting as electricity costs often depend linearly on prices but the RMSE is based on squared errors.

epftoolbox.evaluation.RMSE

epftoolbox.evaluation.**RMSE**(*p_real*, *p_pred*)

Function that computes the root mean square error (RMSE) between two forecasts

$$\text{RMSE} = \frac{1}{N}\sum_{i=1}^{N}\sqrt{\left(p_{\text{real}}[i]p_{\text{pred}}[i]\right)^2}$$

`p_real` and `p_pred` can either be of shape $(n_{\text{days}}, n_{\text{prices/day}})$, $(n_{\text{prices}}, 1)$, or $(n_{\text{prices}},)$ where $n_{\text{prices}} = n_{\text{days}} \cdot n_{\text{prices/day}}$.

Parameters

- **p_real** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the real prices.
- **p_pred** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the predicted prices.

Returns The root mean square error (RMSE).

Return type float

Example

```
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
>>> forecast = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
...                         'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
>>>
>>> # Transforming indices to datetime format
>>> forecast.index = pd.to_datetime(forecast.index)
>>>
>>> # Reading data from the NP market
>>> _, df_test = read_data(path='.', dataset='NP', begin_test_date=forecast.
↪index[0],
...                         end_test_date=forecast.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
>>>
>>> # Extracting forecast of DNN ensemble and display
>>> fc_DNN_ensemble = forecast.loc[:, ['DNN Ensemble']]
>>>
>>> # Extracting real price and display
>>> real_price = df_test.loc[:, ['Price']]
>>>
>>> # Building the same datasets with shape (ndays, n_prices/day) instead
```

(continues on next page)

(continued from previous page)

```
>>> # of shape (nprices, 1) and display
>>> fc_DNN_ensemble_2D = pd.DataFrame(fc_DNN_ensemble.values.reshape(-1, 24),
...                                   index=fc_DNN_ensemble.index[::24],
...                                   columns=['h' + str(hour) for hour in␣
↪range(24)])
>>> real_price_2D = pd.DataFrame(real_price.values.reshape(-1, 24),
...                              index=real_price.index[::24],
...                              columns=['h' + str(hour) for hour in range(24)])
>>> fc_DNN_ensemble_2D.head()
                  h0         h1         h2  ...        h21        h22        h23
2016-12-27  24.349676  23.127774  22.208617  ...  27.686771  27.045763  25.724071
2016-12-28  25.453866  24.707317  24.452384  ...  29.424558  28.627130  27.321902
2016-12-29  28.209516  27.715400  27.182692  ...  28.473288  27.926241  27.153401
2016-12-30  28.002935  27.467572  27.028558  ...  29.086532  28.518688  27.738548
2016-12-31  25.732282  24.668331  23.951569  ...  26.965008  26.450995  25.637346
```

According to the paper, the RMSE of the DNN ensemble for the NP market is 3.333. Let's test the metric for different conditions

```
>>> # Evaluating RMSE when real price and forecasts are both dataframes
>>> RMSE(p_pred=fc_DNN_ensemble, p_real=real_price)
3.3331928060389995
>>>
>>> # Evaluating RMSE when real price and forecasts are both numpy arrays
>>> RMSE(p_pred=fc_DNN_ensemble.values, p_real=real_price.values)
3.3331928060389995
>>>
>>> # Evaluating RMSE when input values are of shape (ndays, n_prices/day) instead
>>> # of shape (nprices, 1)
>>> # Dataframes
>>> RMSE(p_pred=fc_DNN_ensemble_2D, p_real=real_price_2D)
3.3331928060389995
>>> # Numpy arrays
>>> RMSE(p_pred=fc_DNN_ensemble_2D.values, p_real=real_price_2D.values)
3.3331928060389995
>>>
>>> # Evaluating RMSE when input values are of shape (nprices,)
>>> # instead of shape (nprices, 1)
>>> # Pandas Series
>>> RMSE(p_pred=fc_DNN_ensemble.loc[:, 'DNN Ensemble'],
...      p_real=real_price.loc[:, 'Price'])
3.3331928060389995
>>> # Numpy arrays
>>> RMSE(p_pred=fc_DNN_ensemble.values.squeeze(),
...      p_real=real_price.values.squeeze())
3.3331928060389995
```

4.1.3 MAPE

Another popular metric for electricity price forecasting is the mean absolute percentage error (MAPE):

$$\text{MAPE} = \frac{1}{N}\sum_{k=1}^{N}\frac{|p_k - \hat{p}_k|}{|p_k|}. \quad (4.5)$$

This metric computes the *MAE* between the predicted prices and the real prices and normalizes it by the absolute value of the real prices.

While it provides a relative error metric that would grant comparison between datasets, its values become very large with prices close to zero (regardless of the actual absolute errors) and is also not very informative.

epftoolbox.evaluation.MAPE

`epftoolbox.evaluation.`**`MAPE`**(*p_real*, *p_pred*, *noNaN=False*)

Function that computes the mean absolute percentage error (MAPE) between two forecasts:

$$\text{MAPE} = \frac{1}{N}\sum_{i=1}^{N}\frac{\left|p_{\text{real}}[i]p_{\text{pred}}[i]\right|}{\left|Y_{\text{real}}[i]\right|}$$

`p_real` and `p_pred` can either be of shape $(n_{\text{days}}, n_{\text{prices/day}})$, $(n_{\text{prices}}, 1)$, or $(n_{\text{prices}},)$ where $n_{\text{prices}} = n_{\text{days}} \cdot n_{\text{prices/day}}$.

Parameters

- **p_real** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the real prices.
- **p_pred** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the predicted prices.
- **noNaN** (*bool, optional*) – Bool to remove the NaN values resulting of dividing by 0 in the MAPE. It has to be used if any value in p_real is 0.

Returns The mean absolute percentage error (MAPE).

Return type float

Example

```
>>> from epftoolbox.evaluation import MAPE
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
>>> forecast = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
...                         'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
>>>
>>> # Transforming indices to datetime format
>>> forecast.index = pd.to_datetime(forecast.index)
>>>
>>> # Reading data from the NP market
>>> _, df_test = read_data(path='.', dataset='NP', begin_test_date=forecast.
↪index[0],
...                         end_test_date=forecast.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Extracting forecast of DNN ensemble and display
>>> fc_DNN_ensemble = forecast.loc[:, ['DNN Ensemble']]
>>>
>>> # Extracting real price and display
>>> real_price = df_test.loc[:, ['Price']]
>>>
>>> # Building the same datasets with shape (ndays, n_prices/day) instead
>>> # of shape (nprices, 1) and display
>>> fc_DNN_ensemble_2D = pd.DataFrame(fc_DNN_ensemble.values.reshape(-1, 24),
...                                   index=fc_DNN_ensemble.index[::24],
...                                   columns=['h' + str(hour) for hour in␣
↪range(24)])
>>> real_price_2D = pd.DataFrame(real_price.values.reshape(-1, 24),
...                              index=real_price.index[::24],
...                              columns=['h' + str(hour) for hour in range(24)])
>>> fc_DNN_ensemble_2D.head()
                  h0         h1         h2  ...        h21        h22        h23
2016-12-27  24.349676  23.127774  22.208617  ...  27.686771  27.045763  25.724071
2016-12-28  25.453866  24.707317  24.452384  ...  29.424558  28.627130  27.321902
2016-12-29  28.209516  27.715400  27.182692  ...  28.473288  27.926241  27.153401
2016-12-30  28.002935  27.467572  27.028558  ...  29.086532  28.518688  27.738548
2016-12-31  25.732282  24.668331  23.951569  ...  26.965008  26.450995  25.637346
```

According to the paper, the MAPE of the DNN ensemble for the NP market is 5.38%. Let's test the metric for different conditions

```
>>> # Evaluating MAPE when real price and forecasts are both dataframes
>>> MAPE(p_pred=fc_DNN_ensemble, p_real=real_price) * 100
5.376051161768693
>>>
>>> # Evaluating MAPE when real price and forecasts are both numpy arrays
>>> MAPE(p_pred=fc_DNN_ensemble.values, p_real=real_price.values) * 100
5.376051161768693
>>>
>>> # Evaluating MAPE when input values are of shape (ndays, n_prices/day) instead
>>> # of shape (nprices, 1)
>>> # Dataframes
>>> MAPE(p_pred=fc_DNN_ensemble_2D, p_real=real_price_2D) * 100
5.376051161768693
>>> # Numpy arrays
>>> MAPE(p_pred=fc_DNN_ensemble_2D.values, p_real=real_price_2D.values) * 100
5.376051161768693
>>>
>>> # Evaluating MAPE when input values are of shape (nprices,)
>>> # instead of shape (nprices, 1)
>>> # Pandas Series
>>> MAPE(p_pred=fc_DNN_ensemble.loc[:, 'DNN Ensemble'],
...      p_real=real_price.loc[:, 'Price']) * 100
5.376051161768693
>>> # Numpy arrays
>>> MAPE(p_pred=fc_DNN_ensemble.values.squeeze(),
...      p_real=real_price.values.squeeze()) * 100
5.376051161768693
```

4.1.4 sMAPE

Another popular metric in the field of electricity price forecasting is the symmetric mean absolute percentage error (sMAPE):

$$\text{sMAPE} = \frac{1}{N}\sum_{k=1}^{N} 2\frac{|p_k - \hat{p}_k|}{|p_k| + |\hat{p}_k|}, \tag{4.6}$$

This metric computes the *MAE* between the predicted prices and the real prices and normalizes it by the average of the absolute value of both quantities. Note, that there are multiple versions of sMAPE and here we consider the most sensible one.

Although the sMAPE provides a metric based on relative errors that would grant comparison between datasets and even though it solves some of the issues of *MAE*, *RMSE*, and *MAPE* and, it has a statistical distribution with undefined mean and infinite variance.

epftoolbox.evaluation.sMAPE

`epftoolbox.evaluation.`**`sMAPE`**(*p_real*, *p_pred*)

Function that computes the symmetric mean absolute percentage error (sMAPE) between two forecasts

Note that there are multiple versions of sMAPE, here we con-sider the most sensible one :

$$\text{sMAPE} = \frac{1}{N}\sum_{i=1}^{N} \frac{2\left|p_{\text{real}}[i]p_{\text{pred}}[i]\right|}{\left|P_{\text{real}}[i]\right| + \left|P_{\text{pred}}[i]\right|}$$

`p_real` and `p_pred` can either be of shape $(n_{\text{days}}, n_{\text{prices/day}})$, $(n_{\text{prices}}, 1)$, or $(n_{\text{prices}},)$ where $n_{\text{prices}} = n_{\text{days}} \cdot n_{\text{prices/day}}$.

Parameters

- **p_real** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the real prices.
- **p_pred** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the predicted prices.

Returns The symmetric mean absolute percentage error (sMAPE).

Return type float

Example

```
>>> from epftoolbox.evaluation import sMAPE
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
```

(continues on next page)

(continued from previous page)

```
>>> forecast = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
...                         'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
>>>
>>> # Transforming indices to datetime format
>>> forecast.index = pd.to_datetime(forecast.index)
>>>
>>> # Reading data from the NP market
>>> _, df_test = read_data(path='.', dataset='NP', begin_test_date=forecast.
↪index[0],
...                         end_test_date=forecast.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
>>>
>>> # Extracting forecast of DNN ensemble and display
>>> fc_DNN_ensemble = forecast.loc[:, ['DNN Ensemble']]
>>>
>>> # Extracting real price and display
>>> real_price = df_test.loc[:, ['Price']]
>>>
>>> # Building the same datasets with shape (ndays, n_prices/day) instead
>>> # of shape (nprices, 1) and display
>>> fc_DNN_ensemble_2D = pd.DataFrame(fc_DNN_ensemble.values.reshape(-1, 24),
...                                   index=fc_DNN_ensemble.index[::24],
...                                   columns=['h' + str(hour) for hour in␣
↪range(24)])
>>> real_price_2D = pd.DataFrame(real_price.values.reshape(-1, 24),
...                              index=real_price.index[::24],
...                              columns=['h' + str(hour) for hour in range(24)])
>>> fc_DNN_ensemble_2D.head()
                  h0         h1         h2  ...        h21        h22        h23
2016-12-27  24.349676  23.127774  22.208617  ...  27.686771  27.045763  25.724071
2016-12-28  25.453866  24.707317  24.452384  ...  29.424558  28.627130  27.321902
2016-12-29  28.209516  27.715400  27.182692  ...  28.473288  27.926241  27.153401
2016-12-30  28.002935  27.467572  27.028558  ...  29.086532  28.518688  27.738548
2016-12-31  25.732282  24.668331  23.951569  ...  26.965008  26.450995  25.637346
```

According to the paper, the sMAPE of the DNN ensemble for the NP market is 4.85%. Let's test the metric for different conditions

```
>>> # Evaluating sMAPE when real price and forecasts are both dataframes
>>> sMAPE(p_pred=fc_DNN_ensemble, p_real=real_price) * 100
4.846295174735425
>>>
>>> # Evaluating sMAPE when real price and forecasts are both numpy arrays
>>> sMAPE(p_pred=fc_DNN_ensemble.values, p_real=real_price.values) * 100
4.846295174735425
>>>
>>> # Evaluating sMAPE when input values are of shape (ndays, n_prices/day)␣
↪instead
>>> # of shape (nprices, 1)
>>> # Dataframes
>>> sMAPE(p_pred=fc_DNN_ensemble_2D, p_real=real_price_2D) * 100
4.846295174735425
>>> # Numpy arrays
>>> sMAPE(p_pred=fc_DNN_ensemble_2D.values, p_real=real_price_2D.values) * 100
```

(continues on next page)

(continued from previous page)

```
4.846295174735425
>>>
>>> # Evaluating sMAPE when input values are of shape (nprices,)
>>> # instead of shape (nprices, 1)
>>> # Pandas Series
>>> sMAPE(p_pred=fc_DNN_ensemble.loc[:, 'DNN Ensemble'],
...       p_real=real_price.loc[:, 'Price']) * 100
4.846295174735425
>>> # Numpy arrays
>>> sMAPE(p_pred=fc_DNN_ensemble.values.squeeze(),
...       p_real=real_price.values.squeeze()) * 100
4.846295174735425
```

4.1.5 MASE

Considering the errors of standard metrics described in the *introduction*, metrics based on scaled errors, where a scaled error is simply the *MAE* scaled by the in-sample *MAE* of a `naive forecast`, are arguably better. A scaled error has the nice interpretation of being lower/larger than one if it is better/worse than the average naive forecast evaluated in-sample.

A metric based on this concept is the mean absolute scaled error (MASE), and in the context of one-step ahead forecasting is defined as:

$$\text{MASE} = \frac{1}{N}\sum_{k=1}^{N}\frac{|p_k - \hat{p}_k|}{\frac{1}{n-1}\sum_{i=2}^{n}|p_i^{\text{in}} - p_{i-1}^{\text{in}}|}, \quad (4.7)$$

where p_i^{in} is the i^{th} price in the in-sample dataset and n the size of the in-sample dataset. For seasonal time series, the MASE may be defined using the *MAE* of a seasonal naive model in the denominator:

$$\text{MASE}_m = \frac{1}{N}\sum_{k=1}^{N}\frac{|p_k - \hat{p}_k|}{\frac{1}{n-m}\sum_{i=m+1}^{n}|p_i^{\text{in}} - p_{i-m}^{\text{in}}|} \quad (4.8)$$

where m represents the seasonal length (in the case of day-ahead prices that could be either 24 or 168 representing the daily and weekly seasonalities). As an alternative, the `naive forecast` can also be defined on the standard naive forecast for price forecasting (using daily seasonality for Tuesday to Friday and weekly seasonality for Saturday to Monday).

epftoolbox.evaluation.MASE

`epftoolbox.evaluation.`**`MASE`**(*p_real*, *p_pred*, *p_real_in*, *m=None*, *freq='1H'*)

Function that computes the mean absolute scaled error (MASE) between two forecasts:

$$\text{MASE}_{\text{m}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\left|p_{\text{real}}[i] p_{\text{pred}}[i]\right|}{\text{MAE}(p_{\text{real\_in}}, p_{\text{naive\_in}}, m)}.$$

The numerator is the *MAE* of a naive forecast `Ynaive_in` that is built using the insample dataset `p_real_in` and the *naive_forecast* function with a seasonality index `m`.

If the datasets provided are numpy.ndarray objects, the function requires a `freq` argument specifying the data frequency. The `freq` argument must take one of the following four values `'1H'` for 1 hour, `'30T'` for 30 minutes, `'15T'` for 15 minutes, or `'5T'` for 5 minutes, (these are the four standard values in day-ahead electricity markets).

Also, if the datasets provided are numpy.ndarray objects, `m` has to be 24 or 168, i.e. the *naive_forecast* cannot be the standard in electricity price forecasting because the input data does not have associated a day of the week.

`p_real`, `p_pred`, and *p_real_in*' can either be of shape $(n_{\text{days}}, n_{\text{prices/day}})$, $(n_{\text{prices}}, 1)$, or $(n_{\text{prices}},)$ where $n_{\text{prices}} = n_{\text{days}} \cdot n_{\text{prices/day}}$

Parameters

- **p_real** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the real prices.
- **p_pred** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the predicted prices.
- **p_real_in** (*numpy.ndarray, pandas.DataFrame*) – Insample dataset that is used to compute build a *naive_forecast* and compute its *MAE*
- **m** (*int, optional*) – Index that specifies the seasonality in the *naive_forecast* used to compute the normalizing insample MAE. It can be be `'D'` for daily seasonality, `'W'` for weekly seasonality, or None for the standard naive forecast in electricity price forecasting, i.e. daily seasonality for Tuesday to Friday and weekly seasonality for Saturday to Monday.
- **freq** (*str, optional*) – Frequency of the data if `p_real`, `p_pred`, and `p_real_in` are numpy.ndarray objects. It must take one of the following four values `'1H'` for 1 hour, `'30T'` for 30 minutes, `'15T'` for 15 minutes, or `'5T'` for 5 minutes, (these are the four standard values in day-ahead electricity markets).

Returns The mean absolute scaled error (MASE).

Return type float

Example

```
>>> from epftoolbox.evaluation import MASE
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
>>> forecast = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
...                         'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
>>>
>>> # Transforming indices to datetime format
>>> forecast.index = pd.to_datetime(forecast.index)
>>>
>>> # Reading data from the NP market
```

(continues on next page)

(continued from previous page)

```
>>> df_train, df_test = read_data(path='.', dataset='NP', begin_test_
↪date=forecast.index[0],
...                            end_test_date=forecast.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
>>>
>>> # Extracting forecast of DNN ensemble and display
>>> fc_DNN_ensemble = forecast.loc[:, ['DNN Ensemble']]
>>>
>>> # Extracting real price and display
>>> real_price = df_test.loc[:, ['Price']]
>>> real_price_insample = df_train.loc[:, ['Price']]
>>>
>>> # Building the same datasets with shape (ndays, n_prices/day) instead
>>> # of shape (nprices, 1) and display
>>> fc_DNN_ensemble_2D = pd.DataFrame(fc_DNN_ensemble.values.reshape(-1, 24),
...                                   index=fc_DNN_ensemble.index[::24],
...                                   columns=['h' + str(hour) for hour in␣
↪range(24)])
>>> real_price_2D = pd.DataFrame(real_price.values.reshape(-1, 24),
...                              index=real_price.index[::24],
...                              columns=['h' + str(hour) for hour in range(24)])
>>> real_price_insample_2D = pd.DataFrame(real_price_insample.values.reshape(-1,␣
↪24),
...                              index=real_price_insample.index[::24],
...                              columns=['h' + str(hour) for hour in range(24)])
>>>
>>> fc_DNN_ensemble_2D.head()
                  h0         h1         h2  ...        h21        h22        h23
2016-12-27  24.349676  23.127774  22.208617  ...  27.686771  27.045763  25.724071
2016-12-28  25.453866  24.707317  24.452384  ...  29.424558  28.627130  27.321902
2016-12-29  28.209516  27.715400  27.182692  ...  28.473288  27.926241  27.153401
2016-12-30  28.002935  27.467572  27.028558  ...  29.086532  28.518688  27.738548
2016-12-31  25.732282  24.668331  23.951569  ...  26.965008  26.450995  25.637346
>>>
```

Let's test the metric for different conditions.

```
>>> # Evaluating MASE when real price and forecasts are both dataframes
>>> MASE(p_pred=fc_DNN_ensemble, p_real=real_price,
...      p_real_in=real_price_insample, m='W')
0.5217886515713188
>>>
>>> # Evaluating MASE when real price and forecasts are both numpy arrays
>>> MASE(p_pred=fc_DNN_ensemble.values, p_real=real_price.values,
...      p_real_in=real_price_insample.values, m='W', freq='1H')
0.5217886515713188
>>>
>>> # Evaluating MASE when input values are of shape (ndays, n_prices/day) instead
>>> # of shape (nprices, 1)
>>> # Dataframes
>>> MASE(p_pred=fc_DNN_ensemble_2D, p_real=real_price_2D,
...      p_real_in=real_price_insample_2D, m='W')
0.5217886515713188
>>> # Numpy arrays
>>> MASE(p_pred=fc_DNN_ensemble_2D.values, p_real=real_price_2D.values,
...      p_real_in=real_price_insample_2D.values, m='W', freq='1H')
0.5217886515713188
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Evaluating MASE when input values are of shape (nprices,)
>>> # instead of shape (nprices, 1)
>>> # Pandas Series
>>> MASE(p_pred=fc_DNN_ensemble.loc[:, 'DNN Ensemble'],
...      p_real=real_price.loc[:, 'Price'],
...      p_real_in=real_price_insample.loc[:, 'Price'], m='W')
0.5217886515713188
>>> # Numpy arrays
>>> MASE(p_pred=fc_DNN_ensemble.values.squeeze(),
...      p_real=real_price.values.squeeze(),
...      p_real_in=real_price_insample.values.squeeze(), m='W', freq='1H')
0.5217886515713188
```

4.1.6 rMAE

While scaled errors do indeed solve the issues of more traditional metrics, they have other associated problems that make them not unsuitable in the context of EPF:

1. As *MASE* depends on the in-sample dataset, forecasting methods with different calibration windows will naturally have to consider different in-sample datasets. As a result, the *MASE* of each model will be based on a different scaling factor and comparisons between models cannot be drawn.
2. The same argument applies to models with and without rolling windows. The latter will use a different in-sample dataset at every time point while the former will keep the in-sample dataset constant.
3. In ensembles of models with different calibration windows, the *MASE* cannot be defined as the calibration window of the ensemble is undefined.
4. Drawing comparisons across different time series is problematic as electricity prices are not stationary. For example, an in-sample dataset with spikes and an out-of-sample dataset without spikes will lead to a smaller *MASE* than if we consider the same market but with the in-sample/out-sample datasets reversed.

To solve these issues, an arguably better metric is the relative MAE (rMAE). Similar to *MASE*, rMAE normalizes the *MAE* by the *MAE* of a naive forecast. However, instead of considering the in-sample dataset, the naive forecast is built based on the out-of-sample dataset. In the context In the context of one-step ahead forecasting is defined as:

$$\text{rMAE} = \frac{1}{N}\sum_{k=1}^{N}\frac{|p_k - \hat{p}_k|}{\frac{1}{N-1}\sum_{i=2}^{N}|p_i - p_{i-1}|}. \quad (4.9)$$

For seasonal time series, the rMAE may be defined using the *MAE* of a seasonal naive model in the denominator:

$$\text{rMAE}_m = \frac{1}{N}\sum_{k=1}^{N}\frac{|p_k - \hat{p}_k|}{\frac{1}{N-m}\sum_{i=m+1}^{N}|p_i - p_{i-m}|} \quad (4.10)$$

where m represents the seasonal length (in the case of day-ahead prices that could be either 24 or 168 representing the daily and weekly seasonalities). As an alternative, the naive forecast can also be defined on the standard naive forecast for price forecasting (using daily seasonality for Tuesday to Friday and weekly seasonality for Saturday to Monday).

epftoolbox.evaluation.rMAE

epftoolbox.evaluation.**rMAE**(*p_real*, *p_pred*, *m=None*, *freq='1H'*)

Function that computes the relative mean absolute error (rMAE) between two forecasts:

$$\text{rMAE}_\text{m} = \frac{1}{N}\sum_{i=1}^{N}\frac{\left|p_\text{real}[i]p_\text{pred}[i]\right|}{\text{MAE}(p_\text{real}, p_\text{naive})}.$$

The numerator is the *MAE* of a naive forecast `p_naive` that is built using the dataset `p_real` and the *naive_forecast* function with a seasonality index `m`.

If the datasets provided are numpy.ndarray objects, the function requires a `freq` argument specifying the data frequency. The `freq` argument must take one of the following four values `'1H'` for 1 hour, `'30T'` for 30 minutes, `'15T'` for 15 minutes, or `'5T'` for 5 minutes, (these are the four standard values in day-ahead electricity markets).

Also, if the datasets provided are numpy.ndarray objects, `m` has to be `'D'` or `'W'`, i.e. the *naive_forecast* cannot be the standard in electricity price forecasting because the input data does not have associated a day of the week.

`p_real`, `p_pred`, and *p_real_in*' can either be of shape $(n_\text{days}, n_\text{prices/day})$, $(n_\text{prices}, 1)$, or $(n_\text{prices},)$ where $n_\text{prices} = n_\text{days} \cdot n_\text{prices/day}$

Parameters

- **p_real** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the real prices.
- **p_pred** (*numpy.ndarray, pandas.DataFrame*) – Array/dataframe containing the predicted prices.
- **m** (*int, optional*) – Index that specifies the seasonality in the *naive_forecast* used to compute the normalizing insample *MAE*. It can be be `'D'` for daily seasonality, `'W'` for weekly seasonality, or None for the standard naive forecast in electricity price forecasting, i.e. daily seasonality for Tuesday to Friday and weekly seasonality for Saturday to Monday.
- **freq** (*str, optional*) – Frequency of the data if `p_real`, `p_pred`, and `p_real_in` are numpy.ndarray objects. It must take one of the following four values `'1H'` for 1 hour, `'30T'` for 30 minutes, `'15T'` for 15 minutes, or `'5T'` for 5 minutes, (these are the four standard values in day-ahead electricity markets). If the shape of `p_real` is (ndays, n_prices_day), freq should be the frequency of the columns not the daily frequency of the rows.

Returns The mean absolute scaled error (MASE).

Return type float

Example

```
>>> from epftoolbox.evaluation import rMAE
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
>>> forecast = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
```

(continues on next page)

(continued from previous page)

```
...                         'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
>>>
>>> # Transforming indices to datetime format
>>> forecast.index = pd.to_datetime(forecast.index)
>>>
>>> # Reading data from the NP market
>>> _, df_test = read_data(path='.', dataset='NP', begin_test_date=forecast.
↪index[0],
...                         end_test_date=forecast.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
>>>
>>> # Extracting forecast of DNN ensemble and display
>>> fc_DNN_ensemble = forecast.loc[:, ['DNN Ensemble']]
>>>
>>> # Extracting real price and display
>>> real_price = df_test.loc[:, ['Price']]
>>>
>>> # Building the same datasets with shape (ndays, n_prices/day) instead
>>> # of shape (nprices, 1) and display
>>> fc_DNN_ensemble_2D = pd.DataFrame(fc_DNN_ensemble.values.reshape(-1, 24),
...                                   index=fc_DNN_ensemble.index[::24],
...                                   columns=['h' + str(hour) for hour in␣
↪range(24)])
>>> real_price_2D = pd.DataFrame(real_price.values.reshape(-1, 24),
...                              index=real_price.index[::24],
...                              columns=['h' + str(hour) for hour in range(24)])
>>> fc_DNN_ensemble_2D.head()
                  h0         h1         h2  ...        h21        h22        h23
2016-12-27  24.349676  23.127774  22.208617  ...  27.686771  27.045763  25.724071
2016-12-28  25.453866  24.707317  24.452384  ...  29.424558  28.627130  27.321902
2016-12-29  28.209516  27.715400  27.182692  ...  28.473288  27.926241  27.153401
2016-12-30  28.002935  27.467572  27.028558  ...  29.086532  28.518688  27.738548
2016-12-31  25.732282  24.668331  23.951569  ...  26.965008  26.450995  25.637346
>>>
```

According to the paper, the rMAE of the DNN ensemble for the NP market is 0.403 when `m='W'`. Let's test the metric for different conditions

```
>>> # Evaluating rMAE when real price and forecasts are both dataframes
>>> rMAE(p_pred=fc_DNN_ensemble, p_real=real_price)
0.5265639198107801
>>>
>>> # Evaluating rMAE when real price and forecasts are both numpy arrays
>>> rMAE(p_pred=fc_DNN_ensemble.values, p_real=real_price.values, m='W', freq='1H
↪')
0.4031805447246898
>>>
>>> # Evaluating rMAE when input values are of shape (ndays, n_prices/day) instead
>>> # of shape (nprices, 1)
>>> # Dataframes
>>> rMAE(p_pred=fc_DNN_ensemble_2D, p_real=real_price_2D, m='W')
0.4031805447246898
>>> # Numpy arrays
>>> rMAE(p_pred=fc_DNN_ensemble_2D.values, p_real=real_price_2D.values, m='W',␣
↪freq='1H')
```

(continues on next page)

(continued from previous page)

```
0.4031805447246898
>>>
>>> # Evaluating rMAE when input values are of shape (nprices,)
>>> # instead of shape (nprices, 1)
>>> # Pandas Series
>>> rMAE(p_pred=fc_DNN_ensemble.loc[:, 'DNN Ensemble'],
...      p_real=real_price.loc[:, 'Price'], m='W')
0.4031805447246898
>>> # Numpy arrays
>>> rMAE(p_pred=fc_DNN_ensemble.values.squeeze(),
...      p_real=real_price.values.squeeze(), m='W', freq='1H')
0.4031805447246898
```

In addition, it also includes an implementation of the standard naive forecasts in electricity price forecasting. These forecasts are used to compute the *MASE* and *rMAE* metrics:

4.1.7 Naive forecast

To compute the *rMAE* and the *MASE*, a naive forecast is employed. The naive forecast can be built by three methods:

1. Considering daily seasonality and assuming that the prices from one day to the other do not change.
2. Considering weekly seasonality and assuming that the prices from one week to the other do not change
3. Considering different seasonality dependening on the day of the week: daily seasonality for Tuesday to Friday and weekly seasonality for Saturday to Monday.

epftoolbox.evaluation.**naive_forecast**(*p_real*, *m=None*, *n_prices_day=24*)

Function to buil the naive forecast for electricity price forecasting

The function is used to compute the accuracy metrics MASE and RMAE

Parameters

- **p_real** (*pandas.DataFrame*) – Dataframe containing the real prices. It must be of shape $(n_{\text{prices}}, 1)$,
- **m** (*int, optional*) – Index that specifies the seasonality in the naive forecast. It can be `'D'` for daily seasonality, `'W'` for weekly seasonality, or `None` for the standard naive forecast in electricity price forecasting, i.e. daily seasonality for Tuesday to Friday and weekly seasonality for Saturday to Monday.
- **n_prices_day** (*int, optional*) – Number of prices in a day. Usually this value is 24 for most day-ahead markets

Returns Dataframe containing the predictions of the naive forecast.

Return type pandas.DataFrame

Standard Metrics

In the field of electricity price forecasting, two of the most widely used accuracy metrics are the *mean absolute error*

(MAE) and the *root mean square error (RMSE)*:

$$\text{MAE} = \frac{1}{N}\sum_{k=1}^{N}|p_k - \hat{p}_k|, \tag{4.11}$$

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(p_k - \hat{p}_k)^2}, \tag{4.12}$$

(4.13)

where p_k and $\hat{p}_k$ respectively represent the real and forecasted prices at time step k.

Despite their popularity, the *MAE* and *RMSE* are not always very informative as absolute errors are hard to compare between different datasets. In addition, *RMSE* has the extra disadvantage of not representing accurately the underlying problem (electricity costs often depend linearly on prices but *RMSE* is based on squared errors).

Another standard metric is the *mean absolute percentage error (MAPE)*:

$$\text{MAPE} = \frac{1}{N}\sum_{k=1}^{N}\frac{|p_k - \hat{p}_k|}{|p_k|}. \tag{4.14}$$

While it provides a relative error metric that would grant comparison between datasets, its values become very large with prices close to zero (regardless of the actual absolute errors) and is also not very informative.

Another propular metric is the *symmetric mean absolute percentage error (sMAPE)*:

$$\text{sMAPE} = \frac{1}{N}\sum_{k=1}^{N}2\frac{|p_k - \hat{p}_k|}{|p_k| + |\hat{p}_k|}, \tag{4.15}$$

Although the *sMAPE* solves some of these issues, it has a statistical distribution with undefined mean and infinite variance.

MASE

Arguably better metrics are those based on scaled errors, where a scaled error is simply the *MAE* scaled by the in-sample *MAE* of a naive forecast. A scaled error has the nice interpretation of being lower/larger than one if it is better/worse than the average naive forecast evaluated in-sample. A metric based on this concept is the *mean absolute scaled error (MASE)*, and in the context of one-step ahead forecasting is defined as:

$$\text{MASE} = \frac{1}{N}\sum_{k=1}^{N}\frac{|p_k - \hat{p}_k|}{\frac{1}{n-1}\sum_{i=2}^{n}|p_i^{\text{in}} - p_{i-1}^{\text{in}}|}, \tag{4.16}$$

where p_i^{in} is the i^{th} price in the in-sample dataset and n the size of the in-sample dataset.

rMAE

While scaled errors do indeed solve the issues of more traditional metrics, they have other associated problems that make them not unsuitable in the context of EPF:

1. As *MASE* depends on the in-sample dataset, forecasting methods with different calibration windows consider different in-sample datasets. Hence, the *MASE* of each model is based on a different scaling factor and comparisons between models cannot be drawn.
2. Drawing comparisons across different time series is problematic as electricity prices are not stationary. For example, an in-sample dataset with spikes and an out-of-sample dataset without spikes will lead to a smaller *MASE* than if we consider the same market but with the in-sample/out-sample datasets reversed.

To solve these issues, an arguably better metric is the *relative MAE (rMAE)* . Similar to *MASE*, *rMAE* normalizes the *MAE* by the *MAE* of a naive forecast. However, instead of considering the in-sample dataset, the naive forecast is built based on the out-of-sample dataset. In the context of one-step ahead forecasting is defined as:

$$\text{rMAE} = \frac{1}{N}\sum_{k=1}^{N}\frac{|p_k - \hat{p}_k|}{\frac{1}{N-1}\sum_{i=2}^{N}|p_i - p_{i-1}|}. \quad (4.17)$$

4.2 Statistical testing

While using adequate metrics to compare the accuracy of the forecasts is important, it is also necessary to analyze whether any difference in accuracy is statistically significant. This is paramount to conclude whether the difference in accuracy does really exist and it is not simply due to random differences between the forecasts.

4.2.1 Diebold-Mariano test

This module provides a function `DM` that implements the one-sided version of the Diebold-Mariano (DM) test in the context of electricity price forecasting.

Besides the DM test, the module also provides a function *`plot_multivariate_DM_test`* to plot the DM results when comparing multiple forecasts.

DM test

The Diebold-Mariano (DM) test is probably the most commonly used tool to evaluate the significance of differences in forecasting accuracy. It is an asymptotic z-test of the hypothesis that the mean of the loss differential series:

$$\Delta_k^{\text{A,B}} = L(\varepsilon_k^{\text{A}}) - L(\varepsilon_k^{\text{B}}) \quad (4.18)$$

where $\varepsilon_k^{\text{Z}} = p_k - \hat{p}_k$ is the prediction error of model Z for time step k and $L(\cdot)$ is the loss function. For point forecasts, we usually take $L(\varepsilon_k^{\text{Z}}) = |\varepsilon_k^{\text{Z}}|^p$ with $p = 1$ or 2, which corresponds to the absolute and squared losses.

This module implements the one-sided version of the DM test using the a function `DM` function. Given the forecast of a model A and the forecast of a model B, the test evaluates the null hypothesis H_0 of the mean of the loss differential of model A being lower or equal than that of model B. Hence, rejecting the null H_0 means that the forecasts of model B are significantly more accurate than those of model A.

The module provides the two standard versions of the test in electricity price forecasting: an univariate and a multivariate version. The univariate version of the test has the advantage of providing a deeper analysis as it indicates which forecast is significantly better for which hour of the days. The multivariate version grants a better representation of the results as it summarizes the comparison in a single p-value.

`epftoolbox.evaluation.`**`DM`**(*p_real*, *p_pred_1*, *p_pred_2*, *norm=1*, *version='univariate'*)

Function that performs the one-sided DM test in the contex of electricity price forecasting

The test compares whether there is a difference in predictive accuracy between two forecast `p_pred_1` and `p_pred_2`. Particularly, the one-sided DM test evaluates the null hypothesis H0 of the forecasting errors of `p_pred_2` being larger (worse) than the forecasting errors `p_pred_1` vs the alternative hypothesis H1 of the errors of `p_pred_2` being smaller (better). Hence, rejecting H0 means that the forecast `p_pred_2` is significantly more accurate that forecast `p_pred_1`. (Note that this is an informal definition. For a formal one we refer to here)

Two versions of the test are possible:

1. A univariate version with as many independent tests performed as prices per day, i.e. 24 tests in most day-ahead electricity markets.

2. A multivariate with the test performed jointly for all hours using the multivariate loss differential series (see this article for details.

Parameters

- **`p_real`** (*`numpy.ndarray`*) – Array of shape $(n_{\text{days}}, n_{\text{prices/day}})$ representing the real market prices
- **`p_pred_1`** (*`TYPE`*) – Array of shape $(n_{\text{days}}, n_{\text{prices/day}})$ representing the first forecast
- **`p_pred_2`** (*`TYPE`*) – Array of shape $(n_{\text{days}}, n_{\text{prices/day}})$ representing the second forecast
- **`norm`** (*`int, optional`*) – Norm used to compute the loss differential series. At the moment, this value must either be 1 (for the norm-1) or 2 (for the norm-2).
- **`version`** (*`str, optional`*) – Version of the test as defined in here. It can have two values: `'univariate` or `'multivariate`

Returns The p-value after performing the test. It is a float in the case of the multivariate test and a numpy array with a p-value per hour for the univariate test

Return type float, numpy.ndarray

Example

```
>>> from epftoolbox.evaluation import DM
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Generating forecasts of multiple models
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
```

(continues on next page)

(continued from previous page)

```
>>> forecasts = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
...                          'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
>>>
>>> # Deleting the real price field as it the actual real price and not a forecast
>>> del forecasts['Real price']
>>>
>>> # Transforming indices to datetime format
>>> forecasts.index = pd.to_datetime(forecasts.index)
>>>
>>> # Extracting the real prices from the market
>>> _, df_test = read_data(path='.', dataset='NP', begin_test_date=forecasts.
↪index[0],
...                         end_test_date=forecasts.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
>>>
>>> real_price = df_test.loc[:, ['Price']]
>>>
>>> # Testing the univariate DM version on an ensemble of DNN models versus an␣
↪ensemble
>>> # of LEAR models
>>> DM(p_real=real_price.values.reshape(-1, 24),
...     p_pred_1=forecasts.loc[:, 'LEAR Ensemble'].values.reshape(-1, 24),
...     p_pred_2=forecasts.loc[:, 'DNN Ensemble'].values.reshape(-1, 24),
...     norm=1, version='univariate')
array([9.99999944e-01, 9.97562415e-01, 8.10333949e-01, 8.85201928e-01,
       9.33505978e-01, 8.78116764e-01, 1.70135981e-02, 2.37961920e-04,
       5.52337353e-04, 6.07843340e-05, 1.51249750e-03, 1.70415008e-03,
       4.22319907e-03, 2.32808010e-03, 3.55958698e-03, 4.80663621e-03,
       1.64841032e-04, 4.55829140e-02, 5.86609688e-02, 1.98878375e-03,
       1.04045731e-01, 8.71203187e-02, 2.64266732e-01, 4.06676195e-02])
>>>
>>> # Testing the multivariate DM version
>>> DM(p_real=real_price.values.reshape(-1, 24),
...     p_pred_1=forecasts.loc[:, 'LEAR Ensemble'].values.reshape(-1, 24),
...     p_pred_2=forecasts.loc[:, 'DNN Ensemble'].values.reshape(-1, 24),
...     norm=1, version='multivariate')
0.003005725748326471
```

plot_multivariate_DM_test

The *plot_multivariate_DM_test* provides an easy-to-use interface to plot in a heat map with a chessboard shape the results of using the DM test to compare the forecasts of multiple models. An example of the heat map is provided below in the function example.

epftoolbox.evaluation.**plot_multivariate_DM_test**(*real_price*, *forecasts*, *norm=1*, *title='DM test'*, *savefig=False*, *path=''*)

Plotting the results of comparing forecasts using the multivariate DM test.

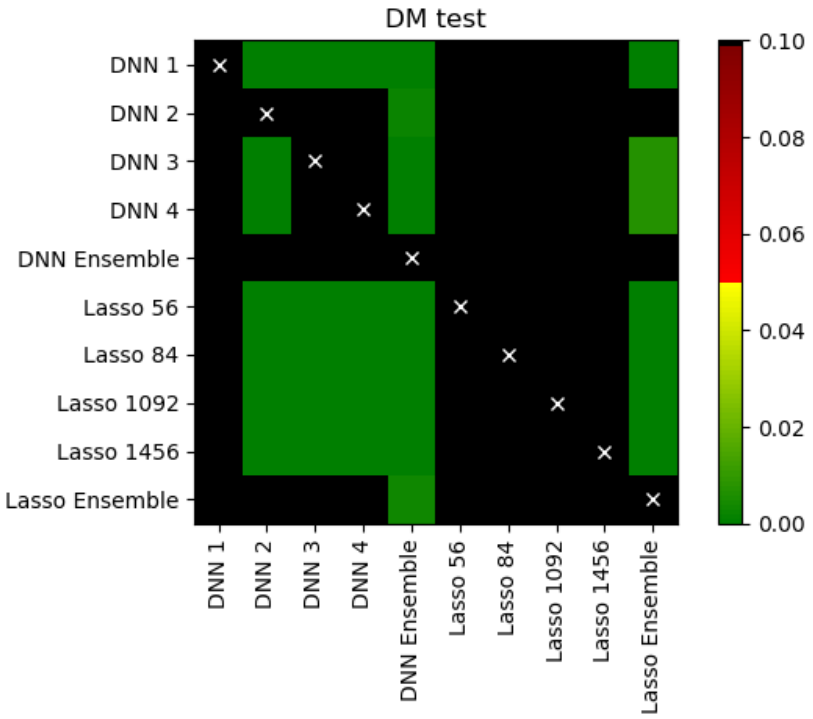
The resulting plot is a heat map in a chessboard shape. It represents the p-value of the null hypothesis of the forecast in the y-axis being significantly more accurate than the forecast in the x-axis. In other words, p-values close to 0 represent cases where the forecast in the x-axis is significantly more accurate than the forecast in the y-axis.

Parameters

- **real_price** (*pandas.DataFrame*) – Dataframe that contains the real prices
- **forecasts** (*TYPE*) – Dataframe that contains the forecasts of different models. The column names are the forecast/model names. The number of datapoints should equal the number of datapoints in `real_price`.
- **norm** (*int, optional*) – Norm used to compute the loss differential series. At the moment, this value must either be 1 (for the norm-1) or 2 (for the norm-2).
- **title** (*str, optional*) – Title of the generated plot
- **savefig** (*bool, optional*) – Boolean that selects whether the figure should be saved in the current folder
- **path** (*str, optional*) – Path to save the figure. Only necessary when *savefig=True*

Example

```
>>> from epftoolbox.evaluation import DM, plot_multivariate_DM_test
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Generating forecasts of multiple models
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
>>> forecasts = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
...                         'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
>>>
>>> # Deleting the real price field as it the actual real price and not a forecast
>>> del forecasts['Real price']
>>>
>>> # Transforming indices to datetime format
>>> forecasts.index = pd.to_datetime(forecasts.index)
>>>
>>> # Extracting the real prices from the market
>>> _, df_test = read_data(path='.', dataset='NP', begin_test_date=forecasts.
↪index[0],
...                         end_test_date=forecasts.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
>>>
>>> real_price = df_test.loc[:, ['Price']]
>>>
>>> # Generating a plot to compare the models using the multivariate DM test
>>> plot_multivariate_DM_test(real_price=real_price, forecasts=forecasts)
```



4.2.2 Giacomini-White test

This module provides a function *GW* that implements the one-sided version of the Giacomini-White (GW) test for Conditional Predictive Ability (CPA) in the context of electricity price forecasting.

Additionally, the module provides a function *plot_multivariate_GW_test* that plots the results of pairwise comparisonsi between multiple models.

GW test

The Giacomini-White (GW) test can be seen as a generalization of the *Diebold-Mariano test* that measures the CPA instead of the Unconditional Predictive Ability. The test, like the *DM variant*, measures the statistical significance of the differences of two models forecasts. It is an asymptotic χ^2-test of the null hypothesis $\mathrm{H}_0 : \phi = 0$ in the regression:

$$\Delta_d^{\mathrm{A,B}} = \phi' \mathbb{X}_{d-1} + \varepsilon_d, \quad (4.19)$$

where $\mathbb{X}_{d-1}$ contains elements from the information set on day $d-1$, i.e., a constant and lags of $\Delta_d^{\mathrm{A,B}}$. The loss differential is obtained like in the DM test:

$$\Delta_k^{\mathrm{A,B}} = L(\varepsilon_k^{\mathrm{A}}) - L(\varepsilon_k^{\mathrm{B}}) \quad (4.20)$$

where $\varepsilon_k^Z = p_k - \hat{p}_k$ is the prediction error of model Z for time step k and $L(\cdot)$ is the loss function. For point forecasts, we usually take $L(\varepsilon_k^Z) = |\varepsilon_k^Z|^p$ with $p = 1$ or 2, which corresponds to the absolute and squared losses.

This module implements the one-sided version of the GW test using the a function *GW* function. Given the forecast of a model A and the forecast of a model B, the test evaluates the null hypothesis H_0 of the CPA of the loss differential of model A being higher or equal than that of model B. Hence, rejecting the null H_0 means that the forecasts of model B are significantly more accurate than those of model A.

The module provides the two standard versions of the test in electricity price forecasting: an univariate and a multivariate version. The univariate version of the test has the advantage of providing a deeper analysis as it indicates which forecast is significantly better for which hour of the days. The multivariate version grants a better representation of the results as it summarizes the comparison in a single p-value.

`epftoolbox.evaluation.`**`GW`**(*p_real*, *p_pred_1*, *p_pred_2*, *norm=1*, *version='univariate'*)

Perform the one-sided GW test

The test compares the Conditional Predictive Accuracy of two forecasts `p_pred_1` and `p_pred_2`. The null H0 is that the CPA of errors `p_pred_1` is higher (better) or equal to the errors of `p_pred_2` vs. the alternative H1 that the CPA of `p_pred_2` is higher. Rejecting H0 means that the forecasts `p_pred_2` are significantly more accurate than forecasts `p_pred_1`. (Note that this is an informal definition. For a formal one we refer here)

Parameters

- **`p_real`** (*`numpy.ndarray`*) – Array of shape $(n_{\text{days}}, n_{\text{prices/day}})$ representing the real market prices
- **`p_pred_1`** (*`TYPE`*) – Array of shape $(n_{\text{days}}, n_{\text{prices/day}})$ representing the first forecast
- **`p_pred_2`** (*`TYPE`*) – Array of shape $(n_{\text{days}}, n_{\text{prices/day}})$ representing the second forecast
- **`norm`** (*`int, optional`*) – Norm used to compute the loss differential series. At the moment, this value must either be 1 (for the norm-1) or 2 (for the norm-2).
- **`version`** (*`str, optional`*) – Version of the test as defined in here. It can have two values: `'univariate'` or `'multivariate'`

Returns The p-value after performing the test. It is a float in the case of the multivariate test and a numpy array with a p-value per hour for the univariate test

Return type float, numpy.ndarray

Example

```
>>> from epftoolbox.evaluation import GW
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Generating forecasts of multiple models
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
>>> forecasts = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
...                         'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
>>>
>>> # Deleting the real price field as it the actual real price and not a forecast
```

(continues on next page)

(continued from previous page)

```
>>> del forecasts['Real price']
>>>
>>> # Transforming indices to datetime format
>>> forecasts.index = pd.to_datetime(forecasts.index)
>>>
>>> # Extracting the real prices from the market
>>> _, df_test = read_data(path='.', dataset='NP', begin_test_date=forecasts.
↪index[0],
...                         end_test_date=forecasts.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
>>>
>>> real_price = df_test.loc[:, ['Price']]
>>>
>>> # Testing the univariate GW version on an ensemble of DNN models versus an␣
↪ensemble
>>> # of LEAR models
>>> GW(p_real=real_price.values.reshape(-1, 24),
...     p_pred_1=forecasts.loc[:, 'LEAR Ensemble'].values.reshape(-1, 24),
...     p_pred_2=forecasts.loc[:, 'DNN Ensemble'].values.reshape(-1, 24),
...     norm=1, version='univariate')
array([1.00000000e+00, 1.00000000e+00, 1.00000000e+00, 1.00000000e+00,
       1.00000000e+00, 1.00000000e+00, 1.03217562e-01, 2.63206239e-03,
       5.23325510e-03, 5.90845414e-04, 6.55116487e-03, 9.85034605e-03,
       3.34250412e-02, 1.80798591e-02, 2.74761848e-02, 3.19436776e-02,
       8.39512169e-04, 2.11907847e-01, 5.79718600e-02, 8.73956638e-03,
       4.30521699e-01, 2.67395381e-01, 6.33448562e-01, 1.99826993e-01])
>>>
>>> # Testing the multivariate GW version
>>> GW(p_real=real_price.values.reshape(-1, 24),
...     p_pred_1=forecasts.loc[:, 'LEAR Ensemble'].values.reshape(-1, 24),
...     p_pred_2=forecasts.loc[:, 'DNN Ensemble'].values.reshape(-1, 24),
...     norm=1, version='multivariate')
0.017598166936843906
```

plot_multivariate_GW_test

The *plot_multivariate_GW_test* provides an easy-to-use interface to plot in a heat map with a chessboard shape the results of using the DM test to compare the forecasts of multiple models. An example of the heat map is provided below in the function example.

epftoolbox.evaluation.**plot_multivariate_GW_test**(*real_price*, *forecasts*, *norm=1*, *title='GW test'*, *savefig=False*, *path=''*)

Plotting the results of comparing forecasts using the multivariate GW test.

The resulting plot is a heat map in a chessboard shape. It represents the p-value of the null hypothesis of the forecast in the y-axis being significantly more accurate than the forecast in the x-axis. In other words, p-values close to 0 represent cases where the forecast in the x-axis is significantly more accurate than the forecast in the y-axis.

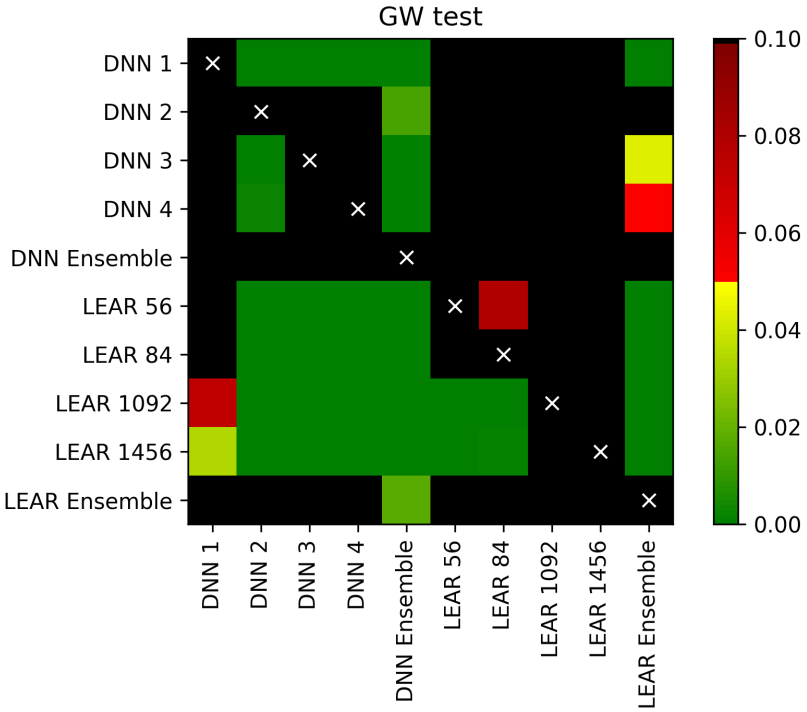
Parameters

- **real_price** (*pandas.DataFrame*) – Dataframe that contains the real prices
- **forecasts** (*TYPE*) – Dataframe that contains the forecasts of different models. The column names are the forecast/model names. The number of datapoints should equal the number of datapoints in `real_price`.

- **norm** (*int, optional*) – Norm used to compute the loss differential series. At the moment, this value must either be 1 (for the norm-1) or 2 (for the norm-2).
- **title** (*str, optional*) – Title of the generated plot
- **savefig** (*bool, optional*) – Boolean that selects whether the figure should be saved in the current folder
- **path** (*str, optional*) – Path to save the figure. Only necessary when *savefig=True*

Example

```
>>> from epftoolbox.evaluation import GW, plot_multivariate_GW_test
>>> from epftoolbox.data import read_data
>>> import pandas as pd
>>>
>>> # Generating forecasts of multiple models
>>>
>>> # Download available forecast of the NP market available in the library␣
↪repository
>>> # These forecasts accompany the original paper
>>> forecasts = pd.read_csv('https://raw.githubusercontent.com/jeslago/epftoolbox/
↪master/' +
...                         'forecasts/Forecasts_NP_DNN_LEAR_ensembles.csv', index_
↪col=0)
>>>
>>> # Deleting the real price field as it the actual real price and not a forecast
>>> del forecasts['Real price']
>>>
>>> # Transforming indices to datetime format
>>> forecasts.index = pd.to_datetime(forecasts.index)
>>>
>>> # Extracting the real prices from the market
>>> _, df_test = read_data(path='.', dataset='NP', begin_test_date=forecasts.
↪index[0],
...                         end_test_date=forecasts.index[-1])
Test datasets: 2016-12-27 00:00:00 - 2018-12-24 23:00:00
>>>
>>> real_price = df_test.loc[:, ['Price']]
>>>
>>> # Generating a plot to compare the models using the multivariate GW test
>>> plot_multivariate_GW_test(real_price=real_price, forecasts=forecasts)
```


GW test
DNN 1
DNN 2
DNN 3
DNN 4
DNN Ensemble
LEAR 56
LEAR 84
LEAR 1092
LEAR 1456
LEAR Ensemble
0.10
0.08
0.06
0.04
0.02
0.00

CHAPTER 5

Examples

The library contains several examples to get the users familiar with the syntax and capabilities of the library. At the moment, the available examples are for using the two state-of-the-art forecasting models: the *LEAR* and the *DNN* models.

For each of the models, there is a simplified example to perform daily recalibration and forecasting, and a more complex, yet flexible, example to perform the same thing. For the DNN, there is also an example for hyperparameter optimization.

5.1 LEAR Examples

This section contains two examples on how to use the *LEAR* model. The *first example* provides an easy-to-use interface for evaluating the LEAR model in a given test dataset. The *second example* provides more flexible interface to perform recalibration and daily forecasting with a LEAR model.

5.1.1 1. Easy recalibration

The first example provides an easy-to-use interface for evaluating the LEAR model in a given test dataset. While this example lacks flexibility, it grants an simple interface to evalute LEAR models in different datasets.

```
"""
Simplified example for using the LEAR model for forecasting prices with daily␣
↪recalibration
"""

# Author: Jesus Lago

# License: AGPL-3.0 License

from epftoolbox.models import evaluate_lear_in_test_dataset
import os
```

(continues on next page)

(continued from previous page)

```
# Market under study. If it not one of the standard ones, the file name
# has to be provided, where the file has to be a csv file
dataset = 'PJM'

# Number of years (a year is 364 days) in the test dataset.
years_test = 2

# Number of days used in the training dataset for recalibration
calibration_window = 364 * 4

# Optional parameters for selecting the test dataset, if either of them is not␣
↪provided,
# the test dataset is built using the years_test parameter. They should either be one␣
↪of
# the date formats existing in python or a string with the following format
# "%d/%m/%Y %H:%M"
begin_test_date = None
end_test_date = None

path_datasets_folder = os.path.join('.', 'datasets')
path_recalibration_folder = os.path.join('.', 'experimental_files')

evaluate_lear_in_test_dataset(path_recalibration_folder=path_recalibration_folder,
                              path_datasets_folder=path_datasets_folder,␣
↪dataset=dataset, years_test=years_test,
                              calibration_window=calibration_window, begin_test_
↪date=begin_test_date,
                              end_test_date=end_test_date)
```

5.1.2 2. Flexible recalibration

The second example provides more flexible interface to perform recalibration and daily forecasting with a LEAR model. While this example is more complex, it grants a flexible interface to use the LEAR model for real-time application.

```
"""
Example for using the LEAR model for forecasting prices with daily recalibration
"""

# Author: Jesus Lago

# License: AGPL-3.0 License

import pandas as pd
import numpy as np
import argparse
import os

from epftoolbox.data import read_data
from epftoolbox.evaluation import MAE, sMAPE
from epftoolbox.models import LEAR

# ------------------------------ EXTERNAL PARAMETERS ---------------------------------
↪---#
```

(continues on next page)

(continued from previous page)

```
parser = argparse.ArgumentParser()

parser.add_argument("--dataset", type=str, default='PJM',
                    help='Market under study. If it not one of the standard ones, the␣
↪file name' +
                         'has to be provided, where the file has to be a csv file')

parser.add_argument("--years_test", type=int, default=2,
                    help='Number of years (a year is 364 days) in the test dataset.␣
↪Used if ' +
                    ' begin_test_date and end_test_date are not provided.')

parser.add_argument("--calibration_window", type=int, default=4 * 364,
                    help='Number of days used in the training dataset for␣
↪recalibration')

parser.add_argument("--begin_test_date", type=str, default=None,
                    help='Optional parameter to select the test dataset. Used in␣
↪combination with ' +
                         'end_test_date. If either of them is not provided, test␣
↪dataset is built ' +
                         'using the years_test parameter. It should either be  a␣
↪string with the ' +
                         ' following format d/m/Y H:M')

parser.add_argument("--end_test_date", type=str, default=None,
                    help='Optional parameter to select the test dataset. Used in␣
↪combination with ' +
                         'begin_test_date. If either of them is not provided, test␣
↪dataset is built ' +
                         'using the years_test parameter. It should either be  a␣
↪string with the ' +
                         ' following format d/m/Y H:M')

args = parser.parse_args()

dataset = args.dataset
years_test = args.years_test
calibration_window = args.calibration_window
begin_test_date = args.begin_test_date
end_test_date = args.end_test_date

path_datasets_folder = os.path.join('.', 'datasets')
path_recalibration_folder = os.path.join('.', 'experimental_files')


# Defining train and testing data
df_train, df_test = read_data(dataset=dataset, years_test=years_test, path=path_
↪datasets_folder,
                              begin_test_date=begin_test_date, end_test_date=end_test_
↪date)

# Defining unique name to save the forecast
forecast_file_name = 'fc_nl' + '_dat' + str(dataset) + '_YT' + str(years_test) + \
                     '_CW' + str(calibration_window) + '.csv'
```

(continues on next page)

(continued from previous page)

```
forecast_file_path = os.path.join(path_recalibration_folder, forecast_file_name)


# Defining empty forecast array and the real values to be predicted in a more␣
↪friendly format
forecast = pd.DataFrame(index=df_test.index[::24], columns=['h' + str(k) for k in␣
↪range(24)])
real_values = df_test.loc[:, ['Price']].values.reshape(-1, 24)
real_values = pd.DataFrame(real_values, index=forecast.index, columns=forecast.
↪columns)

forecast_dates = forecast.index

model = LEAR(calibration_window=calibration_window)

# For loop over the recalibration dates
for date in forecast_dates:

    # For simulation purposes, we assume that the available data is
    # the data up to current date where the prices of current date are not known
    data_available = pd.concat([df_train, df_test.loc[:date + pd.Timedelta(hours=23),␣
↪:]], axis=0)

    # We set the real prices for current date to NaN in the dataframe of available␣
↪data
    data_available.loc[date:date + pd.Timedelta(hours=23), 'Price'] = np.NaN

    # Recalibrating the model with the most up-to-date available data and making a␣
↪prediction
    # for the next day
    Yp = model.recalibrate_and_forecast_next_day(df=data_available, next_day_
↪date=date,
                                                 calibration_window=calibration_
↪window)
    # Saving the current prediction
    forecast.loc[date, :] = Yp

    # Computing metrics up-to-current-date
    mae = np.mean(MAE(forecast.loc[:date].values.squeeze(), real_values.loc[:date].
↪values))
    smape = np.mean(sMAPE(forecast.loc[:date].values.squeeze(), real_values.
↪loc[:date].values)) * 100

    # Pringint information
    print('{} - sMAPE: {:.2f}%  |  MAE: {:.3f}'.format(str(date)[:10], smape, mae))

    # Saving forecast
    forecast.to_csv(forecast_file_path)
```

5.2 DNN Examples

This section contains three examples on how to use the *DNN* model. The *first example* introduces the hyperparameter optimization and feature selection procedure of the *DNN* model. The *second example* provides an easy-to-use interface for evaluating the DNN model in a given test dataset. The *third example* provides more flexible interface to perform recalibration and daily forecasting with a DNN model.

5.2.1 1. Hyperparameter optimization

The first example introduces the hyperparameter optimization and feature selection procedure of the *DNN* model.

```
"""
Example for optimizing the hyperparameter and features of the DNN model
"""

# Author: Jesus Lago

# License: AGPL-3.0 License


from epftoolbox.models import hyperparameter_optimizer

# Number of layers in DNN
nlayers = 2

# Market under study. If it not one of the standard ones, the file name
# has to be provided, where the file has to be a csv file
dataset = 'NP'

# Number of years (a year is 364 days) in the test dataset.
years_test = 2

# Optional parameters for selecting the test dataset, if either of them is not␣
↪provided,
# the test dataset is built using the years_test parameter. They should either be one␣
↪of
# the date formats existing in python or a string with the following format
# "%d/%m/%Y %H:%M"
begin_test_date = None
end_test_date = None

# Boolean that selects whether the validation and training datasets are shuffled
shuffle_train = 1

# Boolean that selects whether a data augmentation technique for DNNs is used
data_augmentation = 0

# Boolean that selects whether we start a new hyperparameter optimization or we␣
↪restart an existing one
new_hyperopt = 1

# Number of years used in the training dataset for recalibration
calibration_window = 4

# Unique identifier to read the trials file of hyperparameter optimization
experiment_id = 1

# Number of iterations for hyperparameter optimization
max_evals = 1500

path_datasets_folder = "./datasets/"
path_hyperparameters_folder = "./experimental_files/"

# Check documentation of the hyperparameter_optimizer for each of the function␣
↪parameters
```

(continues on next page)

(continued from previous page)

```
# In this example, we optimize a model for the PJM market.
# We consider two directories, one for storing the datasets and the other one for the␣
↪experimental files.
# We start a hyperparameter optimization from scratch. We employ 1500 iterations in␣
↪hyperopt,
# 2 years of test data, a DNN with 2 hidden layers, a calibration window of 4 years,
# we avoid data augmentation,  and we provide an experiment_id equal to 1
hyperparameter_optimizer(path_datasets_folder=path_datasets_folder,
                         path_hyperparameters_folder=path_hyperparameters_folder,
                         new_hyperopt=new_hyperopt, max_evals=max_evals,␣
↪nlayers=nlayers, dataset=dataset,
                         years_test=years_test, calibration_window=calibration_window,
                         shuffle_train=shuffle_train, data_augmentation=0, experiment_
↪id=experiment_id,
                         begin_test_date=begin_test_date, end_test_date=end_test_date)
```

5.2.2 2. Easy recalibration

The second example provides an easy-to-use interface for evaluating the LEAR model in a given test dataset. While this example lacks flexibility, it grants an simple interface to evalute LEAR models in different datasets. It is important to note that this example assumes that a hyperparameter optimization procedure has already been performed.

```
"""
Simplified example for using the DNN model for forecasting prices with daily␣
↪recalibration
"""

# Author: Jesus Lago

# License: AGPL-3.0 License

from epftoolbox.models import evaluate_dnn_in_test_dataset
import os

# Number of layers in DNN
nlayers = 2

# Market under study. If it not one of the standard ones, the file name
# has to be provided, where the file has to be a csv file
dataset = 'PJM'

# Number of years (a year is 364 days) in the test dataset.
years_test = 2

# Boolean that selects whether the validation and training datasets were shuffled when
# performing the hyperparameter optimization. Note that it does not select whether
# shuffling is used for recalibration as for recalibration the validation and the
# training datasets are always shuffled.
shuffle_train = 1

# Boolean that selects whether a data augmentation technique for DNNs is used
data_augmentation = 0

# Boolean that selects whether we start a new recalibration or we restart an existing␣
↪one
```

(continues on next page)

(continued from previous page)

```
new_recalibration = 1

# Number of years used in the training dataset for recalibration
calibration_window = 4

# Unique identifier to read the trials file of hyperparameter optimization
experiment_id = 1

# Optional parameters for selecting the test dataset, if either of them is not␣
↪provided,
# the test dataset is built using the years_test parameter. They should either be one␣
↪of
# the date formats existing in python or a string with the following format
# "%d/%m/%Y %H:%M"
begin_test_date = '27/12/2016'
end_test_date = '01/03/2017'

# Set up the paths for saving data (this are the defaults for the library)
path_datasets_folder = os.path.join('.', 'datasets')
path_recalibration_folder = os.path.join('.', 'experimental_files')
path_hyperparameter_folder = os.path.join('.', 'experimental_files')

evaluate_dnn_in_test_dataset(experiment_id, path_hyperparameter_folder=path_
↪hyperparameter_folder,
                             path_datasets_folder=path_datasets_folder, shuffle_
↪train=shuffle_train,
                             path_recalibration_folder=path_recalibration_folder,
                             nlayers=nlayers, dataset=dataset, years_test=years_
↪test,
                             data_augmentation=data_augmentation, calibration_
↪window=calibration_window,
                             new_recalibration=new_recalibration, begin_test_
↪date=begin_test_date,
                             end_test_date=end_test_date)
```

5.2.3 3. Flexible recalibration

The third example provides more flexible interface to perform recalibration and daily forecasting with a LEAR model. While this example is more complex, it grants a flexible interface to use the LEAR model for real-time application. It is important to note that this example assumes that a hyperparameter optimization procedure has already been performed.

```
"""
Example for using the DNN model for forecasting prices with daily recalibration
"""

# Author: Jesus Lago

# License: AGPL-3.0 License

import pandas as pd
import numpy as np
import argparse
import os

from epftoolbox.data import read_data
```

(continues on next page)

(continued from previous page)

```
from epftoolbox.evaluation import MAE, sMAPE
from epftoolbox.models import DNN



# ------------------------------ EXTERNAL PARAMETERS ---------------------------------
↪---#

parser = argparse.ArgumentParser()

parser.add_argument("--nlayers", help="Number of layers in DNN", type=int, default=2)

parser.add_argument("--dataset", type=str, default='PJM',
                    help='Market under study. If it not one of the standard ones, the␣
↪file name' +
                         'has to be provided, where the file has to be a csv file')

parser.add_argument("--years_test", type=int, default=2,
                    help='Number of years (a year is 364 days) in the test dataset.␣
↪Used if ' +
                    ' begin_test_date and end_test_date are not provided.')

parser.add_argument("--shuffle_train", type=int, default=1,
                    help='Boolean that selects whether the validation and training␣
↪datasets were' +
                    ' shuffled when performing the hyperparameter optimization.')

parser.add_argument("--data_augmentation", type=int, default=0,
                    help='Boolean that selects whether a data augmentation technique␣
↪for DNNs is used')

parser.add_argument("--new_recalibration", type=int, default=1,
                    help='Boolean that selects whether we start a new recalibration␣
↪or we restart an' +
                         ' existing one')

parser.add_argument("--calibration_window", type=int, default=4,
                    help='Number of years used in the training dataset for␣
↪recalibration')

parser.add_argument("--experiment_id", type=int, default=1,
                    help='Unique identifier to read the trials file of hyperparameter␣
↪optimization')

parser.add_argument("--begin_test_date", type=str, default=None,
                    help='Optional parameter to select the test dataset. Used in␣
↪combination with ' +
                         'end_test_date. If either of them is not provided, test␣
↪dataset is built ' +
                         'using the years_test parameter. It should either be  a␣
↪string with the ' +
                         ' following format d/m/Y H:M')

parser.add_argument("--end_test_date", type=str, default=None,
                    help='Optional parameter to select the test dataset. Used in␣
↪combination with ' +
                         'begin_test_date. If either of them is not provided, test␣
↪dataset is built ' +
```

(continues on next page)

(continued from previous page)

```
                        'using the years_test parameter. It should either be  a␣
↪string with the ' +
                        ' following format d/m/Y H:M')

args = parser.parse_args()

nlayers = args.nlayers
dataset = args.dataset
years_test = args.years_test
shuffle_train = args.shuffle_train
data_augmentation = args.data_augmentation
new_recalibration = args.new_recalibration
calibration_window = args.calibration_window
experiment_id = args.experiment_id
begin_test_date = args.begin_test_date
end_test_date = args.end_test_date

path_datasets_folder = os.path.join('.', 'datasets')
path_recalibration_folder = os.path.join('.', 'experimental_files')
path_hyperparameter_folder = os.path.join('.', 'experimental_files')

# Defining train and testing data
df_train, df_test = read_data(dataset=dataset, years_test=years_test, path=path_
↪datasets_folder,
                              begin_test_date=begin_test_date, end_test_date=end_test_
↪date)

# Defining unique name to save the forecast
forecast_file_name = 'fc_nl' + str(nlayers) + '_dat' + str(dataset) + \
                     '_YT' + str(years_test) + '_SF' + str(shuffle_train) + \
                     '_DA' * data_augmentation + '_CW' + str(calibration_window) + \
                     '_' + str(experiment_id) + '.csv'

forecast_file_path = os.path.join(path_recalibration_folder, forecast_file_name)

# Defining empty forecast array and the real values to be predicted in a more␣
↪friendly format
forecast = pd.DataFrame(index=df_test.index[::24], columns=['h' + str(k) for k in␣
↪range(24)])
real_values = df_test.loc[:, ['Price']].values.reshape(-1, 24)
real_values = pd.DataFrame(real_values, index=forecast.index, columns=forecast.
↪columns)

# If we are not starting a new recalibration but re-starting an old one, we import the
# existing files and print metrics
if not new_recalibration:
    # Import existinf forecasting file
    forecast = pd.read_csv(forecast_file_path, index_col=0)
    forecast.index = pd.to_datetime(forecast.index)

    # Reading dates to still be forecasted by checking NaN values
    forecast_dates = forecast[forecast.isna().any(axis=1)].index

    # If all the dates to be forecasted have already been forecast, we print␣
↪information
    # and exit the script
    if len(forecast_dates) == 0:
```

(continues on next page)

(continued from previous page)

```
        mae = np.mean(MAE(forecast.values.squeeze(), real_values.values))
        smape = np.mean(sMAPE(forecast.values.squeeze(), real_values.values)) * 100
        print('{} - sMAPE: {:.2f}%  |  MAE: {:.3f}'.format('Final metrics', smape,␣
↪mae))

else:
    forecast_dates = forecast.index

model = DNN(
    experiment_id=experiment_id, path_hyperparameter_folder=path_hyperparameter_
↪folder, nlayers=nlayers,
    dataset=dataset, years_test=years_test, shuffle_train=shuffle_train, data_
↪augmentation=data_augmentation,
    calibration_window=calibration_window)


# For loop over the recalibration dates
for date in forecast_dates:

    # For simulation purposes, we assume that the available data is
    # the data up to current date where the prices of current date are not known
    data_available = pd.concat([df_train, df_test.loc[:date + pd.Timedelta(hours=23),␣
↪:]], axis=0)

    # We extract real prices for current date and set them to NaN in the dataframe of␣
↪available data
    data_available.loc[date:date + pd.Timedelta(hours=23), 'Price'] = np.NaN

    # Recalibrating the model with the most up-to-date available data and making a␣
↪prediction
    # for the next day
    Yp = model.recalibrate_and_forecast_next_day(df=data_available, next_day_
↪date=date)

    # Saving the current prediction
    forecast.loc[date, :] = Yp

    # Computing metrics up-to-current-date
    mae = np.mean(MAE(forecast.loc[:date].values.squeeze(), real_values.loc[:date].
↪values))
    smape = np.mean(sMAPE(forecast.loc[:date].values.squeeze(), real_values.
↪loc[:date].values)) * 100

    # Pringint information
    print('{} - sMAPE: {:.2f}%  |  MAE: {:.3f}'.format(str(date)[:10], smape, mae))

    # Saving forecast
    forecast.to_csv(forecast_file_path)
```

CHAPTER 6

Citation

If you use the epftoolbox in a scientific publication, we would appreciate citations to the following paper:

- Jesus Lago, Grzegorz Marcjasz, Bart De Schutter, Rafał Weron. "Forecasting day-ahead electricity prices: A review of state-of-the-art algorithms, best practices and an open-access benchmark". *Applied Energy* 2021; 293:116983. [https://doi.org/10.1016/j.apenergy.2021.116983{]}(https://doi.org/10.1016/j.apenergy.2021.116983).

Bibtex entry:

```
@article{epftoolbox,
```

title = {Forecasting day-ahead electricity prices: A review of state-of-the-art algorithms, best practices and an open-access benchmark}, journal = {Applied Energy}, volume = {293}, pages = {116983}, year = {2021}, doi = {https://doi.org/10.1016/j.apenergy.2021.116983}, author = {Jesus Lago and Grzegorz Marcjasz and Bart {De Schutter} and Rafał Weron}

}

Index