

---

# **pycast Documentation**

***Release v0.0.10-prealpha***

**Christian Schwarz**

November 29, 2012



# CONTENTS

<b>1</b>	<b>Requirements</b>	<b>3</b>
<b>2</b>	<b>License</b>	<b>5</b>
<b>3</b>	<b>Contributors</b>	<b>7</b>
<b>4</b>	<b>Documentation</b>	<b>9</b>
4.1	pycast.common . . . . .	9
4.2	pycast.common.timeseries . . . . .	10
4.3	TimeSeries . . . . .	10
4.4	pycast Smoothing and Forecasting Methods . . . . .	15
4.5	Error Measures . . . . .	24
4.6	Optimization Methods . . . . .	31
4.7	pycast conventions . . . . .	33
<b>5</b>	<b>Useful Links</b>	<b>35</b>
	<b>Python Module Index</b>	<b>37</b>
	<b>Index</b>	<b>39</b>



pycast aims to provide a python module supporting the basics as well as advanced smoothing and forecasting methods that can be used on time series data.

Examples of pycast can be found in `bin/examples`



# REQUIREMENTS

## **pycast**

- nose >= 1.2.1
- coverage >= 3.5.3

## **Documentation**

- sphinx >= 1.1.3

## **Examples**

- itty >= 0.8.1





# LICENSE

Copyright (c) 2012 Christian Schwarz

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



# CONTRIBUTORS

Christian Schwarz, Felix Leupold



# DOCUMENTATION

## 4.1 pycast.common

`class pycast.common.profileme._ProfileDecorator (filelocation)`  
Bases: object

Decorator class that build a wrapper around any function.

**Warning** The decorator does not take recursive calls into account!

`__call__ (func)`

Returns a wrapped version of the called function.

**Parameters** `func` (*Function*) – Function that should be wrapped.

**Returns** Returns a wrapped version of the called function.

**Return type** Function

`__init__ (filelocation)`

Initializes the ProfileMe decorator.

**Parameters**

- **func** (*Function*) – Function that will be profiles.
- **filelocation** (*String*) – Location for the profiling results.

`pycast.common.profileme.profileMe`  
alias of `_ProfileDecorator`

`pycast.common.helper.linear_interpolation (first, last, steps)`  
Interpolates all missing values using linear interpolation.

**Parameters**

- **first** (*Numeric*) – Start value for the interpolation.
- **last** (*Numeric*) – End Value for the interpolation
- **steps** (*Integer*) – Number of missing values that have to be calculated.

**Returns** Returns a list of floats containing only the missing values.

**Return type** List

## 4.2 pystac.common.timeseries

### 4.2.1 Normalization Levels

A `TimeSeries` instance can be normalized by different time granularity levels. Valid values for normalization levels required by `pystac.common.TimeSeries.normalize()` are stored in `pystac.common.timeseries.NormalizationLevels`.

Those levels include:

- “second”
- “minute”
- “hour”
- “day”
- “week”
- “2week”
- “4week”

### 4.2.2 Fusion Methods

Fusion methods that can be used to fusionate multiple data points within the same time bucket. This might sort the list it is used on. Valid values for fusion methods required by `pystac.common.TimeSeries.normalize()` are stored in `pystac.common.timeseries.FusionMethods`.

Valid fusion methods are:

- “sum”: Sums up all valid values stored in the specific time bucket
- “mean”: Calculates the mean value within the time bucket
- “median”: Calculates the median of the given time bucket values. In the case the number of entries within that bucket is even, the larger of the both values will be chosen as median.

### 4.2.3 Interpolation Methods

Interpolation methods that can be used for interpolation missing time buckets. Valid values for interpolation methods required by `pystac.common.TimeSeries.normalize()` are stored in `pystac.common.timeseries.InterpolationMethods`.

Valid values for interpolation methods are:

- “linear”: Use linear interpolation to calculate the missing values

## 4.3 TimeSeries

```
class pystac.common.timeseries.TimeSeries(isNormalized=False, isSorted=False)
    Bases: object
```

A `TimeSeries` instance stores all relevant data for a real world time series.

**Warning** `TimeSeries` instances are NOT threadsafe.

**\_\_add\_\_** (*otherTimeSeries*)

Creates a new TimeSeries instance containing the data of `self` and `otherTimeSeries`.

**Parameters** **otherTimeSeries** (*TimeSeries*) – TimeSeries instance that will be merged with `self`.

**Returns** Returns a new TimeSeries instance containing the data entries of `self` and `otherTimeSeries`.

**Return type** TimeSeries

**\_\_copy\_\_** ()

Returns a new clone of the TimeSeries.

**Returns** Returns a TimeSeries containing the same data and configuration as `self`.

**Return type** TimeSeries

**\_\_eq\_\_** (*otherTimeSeries*)

Returns if `self` and the other TimeSeries are equal.

**TimeSeries are equal to each other if:**

- they contain the same number of entries
- each data entry in one TimeSeries is also member of the other one.

**Parameters** **otherTimeSeries** (*TimeSeries*) – TimeSeries instance that is compared with `self`.

**Returns** `True` if the TimeSeries objects are equal, `False` otherwise.

**Return type** Boolean

**\_\_getitem\_\_** (*index*)

Returns the item stored at the TimeSeries index-th position.

**Parameters** **index** (*Integer*) – Position of the element that should be returned. Starts at 0

**Returns** Returns a list containing [timestamp, data] lists.

**Return type** List

**Raise** Raises an `IndexError` if the index is out of range.

**\_\_init\_\_** (*isNormalized=False, isSorted=False*)

Initializes the TimeSeries.

**Parameters**

- **isNormalized** (*Boolean*) – Within a normalized TimeSeries, all data points have the same temporal distance to each other. When this is `True`, the memory consumption of the TimeSeries might be reduced. Also algorithms will probably run faster on normalized TimeSeries. This should only be set to `True`, if the TimeSeries is really normalized! TimeSeries normalization can be forced by executing `TimeSeries.normalize()`.
- **isSorted** (*Boolean*) – If all data points added to the time series are added in their ascending temporal order, this should set to `True`.

**\_\_iter\_\_** ()

Returns an iterator that can be used to iterate over the data stored within the TimeSeries.

**Returns** Returns an iterator for the TimeSeries.

**Return type** Iterator

`__len__()`

Returns the number of data entries stored in the TimeSeries.

**Returns** Returns an Integer representing the number on data entries stored within the TimeSeries.

**Return type** Integer

`__setitem__(index, value)`

Sets the item at the index-th position of the TimeSeries.

**Parameters**

- **index** (*Integer*) – Index of the element that should be set.
- **value** (*List*) – A list of the form [timestamp, data]

**Raise** Raises an `IndexError` if the index is out of range.

`__str__()`

Returns a string representation of the TimeSeries.

**Returns**

Returns a string representing the TimeSeries in the format:

“TimeSeries([timestamp, data], [timestamp, data], [timestamp, data])”.

**Return type** String

`add_entry(timestamp, data)`

Adds a new data entry to the TimeSeries.

**Parameters**

- **timestamp** – Time stamp of the data. This has either to be a float representing the UNIX epochs or a string containing a timestamp in the given format.
- **data** (*Numeric*) – Actual data value.

`apply(method)`

Applies the given ForecastingAlgorithm or SmoothingMethod from the `pystac.methods` module to the TimeSeries.

**Parameters** **method** (*BaseMethod*) – Method that should be used with the TimeSeries. For more information about the methods take a look into their corresponding documentation.

**Raise** Raises a `StandardError` when the TimeSeries was not normalized and hte method requires a normalized TimeSeries

`check_normalization()`

Checks, if the TimeSeries is normalized.

**Returns** Returns `True` if all data entries of the TimeSeries have an equal temporal distance, `False` otherwise.

`classmethod convert_epoch_to_timestamp(timestamp, format)`

Converts the given float representing UNIX-epochs into an actual timestamp.

**Parameters**

- **timestamp** (*Float*) – Timestamp as UNIX-epochs.
- **format** (*String*) – Format of the given timestamp. This is used to convert the timestamp from UNIX epochs. For valid examples take a look into the `time.strptime()` documentation.



**Returns** Returns the timestamp as defined in format.

**Return type** String

**classmethod** `convert_timestamp_to_epoch(timestamp, format)`

Converts the given timestamp into a float representing UNIX-epochs.

**Parameters**

- **timestamp** (*String*) – Timestamp in the defined format.
- **format** (*String*) – Format of the given timestamp. This is used to convert the timestamp into UNIX epochs. For valid examples take a look into the `time.strptime()` documentation.

**Returns** Returns an float, representing the UNIX-epochs for the given timestamp.

**Return type** Float

**classmethod** `from_json(json, format=None)`

Creates a new TimeSeries instance from the given json string.

**Parameters**

- **json** (*String*) – JSON string, containing the time series data. This should be a string created by `TimeSeries.to_json()`.
- **format** (*String*) – Format of the given timestamps. This is used to convert the timestamps into UNIX epochs, if set. For valid examples take a look into the `time.strptime()` documentation.

**Returns** Returns a TimeSeries instance containing the data.

**Return type** TimeSeries

**Warning** This is probably an unsafe version! Only use it with JSON strings created by `TimeSeries.to_json()`. All assumptions regarding normalization and sort order will be ignored and set to default.

**classmethod** `from_twodim_list(datalist, format=None)`

Creates a new TimeSeries instance from the data stored inside a two dimensional list.

**Parameters**

- **datalist** (*List*) – List containing multiple iterables with at least two values. The first item will always be used as timestamp in the predefined format, the second represents the value. All other items in those sublists will be ignored.
- **format** (*String*) – Format of the given timestamp. This is used to convert the timestamp into UNIX epochs, if necessary. For valid examples take a look into the `time.strptime()` documentation.

**Returns** Returns a TimeSeries instance containing the data from datalist.

**Return type** TimeSeries

**initialize\_from\_sql\_cursor(sqlcursor)**

Initializes the TimeSeries's data from the given SQL cursor.

You need to set the time stamp format using `TimeSeries.set_timeformat()`.

**Parameters** **sqlcursor** (*SQLCursor*) – Cursor that was holds the SQL result for any given “SELECT timestamp, value, ... FROM ...” SQL query. Only the first two attributes of the SQL result will be used.

**Returns** Returns the number of entries added to the TimeSeries.

**Return type** Integer

**is\_normalized()**

Returns if the TimeSeries is normalized.

**Returns** Returns `True` if the TimeSeries is normalized, `False` otherwise.

**Return type** Boolean

**is\_sorted()**

Returns if the TimeSeries is sorted.

**Returns** Returns `True` if the TimeSeries is sorted ascending, `False` in all other cases.

**Return type** Boolean

**normalize** (*normalizationLevel='minute', fusionMethod='mean', interpolationMethod='linear'*)

Normalizes the TimeSeries data points.

If this function is called, the TimeSeries gets ordered ascending automatically. The new timestamps will represent the center of each time bucket. Within a normalized TimeSeries, the temporal distance between two consecutive data points is constant.

**Parameters**

- **normalizationLevel** (*String*) – Level of normalization that has to be applied. The available normalization levels are defined in `timeseries.NormalizationLevels`.
- **fusionMethod** (*String*) – Normalization method that has to be used if multiple data entries exist within the same normalization bucket. The available methods are defined in `timeseries.FusionMethods`.
- **interpolationMethod** (*String*) – Interpolation method that is used if a data entry at a specific time is missing. The available interpolation methods are defined in `timeseries.InterpolationMethods`.

**Raise** Raises a `ValueError` if a `normalizationLevel`, `fusionMethod` or `interpolationMethod` have an unknown value.

**set\_timeformat** (*format=None*)

Sets the TimeSeries global time format.

**Parameters** **format** (*String*) – Format of the timestamp. This is used to convert the timestamp from UNIX epochs when the TimeSeries gets serialized by `TimeSeries.to_json()` and `TimeSeries.to_gnuplot_datafile()`. For valid examples take a look into the `time.strptime()` documentation.

**sort\_timeseries** (*ascending=True*)

Sorts the data points within the TimeSeries according to their occurrence inline.

**Parameters** **ascending** (*Boolean*) – Determines if the TimeSeries will be ordered ascending or descending. If this is set to descending once, the ordered parameter defined in `TimeSeries.__init__()` will be set to `False` FOREVER.

**Returns** Returns `self` for convenience.

**Return type** TimeSeries

**sorted\_timeseries** (*ascending=True*)

Returns a sorted copy of the TimeSeries, preserving the original one.

As an assumption this new TimeSeries is not ordered anymore if a new value is added.

**Parameters** **ascending** (*Boolean*) – Determines if the TimeSeries will be ordered ascending or descending.

**Returns** Returns a new TimeSeries instance sorted in the requested order.

**Return type** TimeSeries

**to\_gnuplot\_datafile** (*datafilepath*)

Dumps the TimeSeries into a gnuplot compatible data file.

**Parameters** **datafilepath** (*String*) – Path used to create the file. If that file already exists, it will be overwritten!

**Returns** Returns `True` if the data could be written, `False` otherwise.

**Return type** Boolean

**to\_json** ()

Returns a JSON representation of the TimeSeries data.

**Returns** Returns a basestring, containing the JSON representation of the current data stored within the TimeSeries.

**Return type** String

**to\_twodim\_list** ()

Serializes the TimeSeries data into a two dimensional list of [timestamp, value] pairs.

**Returns** Returns a two dimensional list containing [timestamp, value] pairs.

**Return type** List

## 4.4 pystac Smoothing and Forecasting Methods

### 4.4.1 Base Methods

```
class pystac.methods.basemethod.BaseMethod (requiredParameters=[], hasToBeSorted=True,
                                              hasToBeNormalized=True)
```

Bases: object

Baseclass for all smoothing and forecasting methods.

```
__init__ (requiredParameters=[], hasToBeSorted=True, hasToBeNormalized=True)
```

Initializes the BaseMethod.

**Parameters**

- **requiredParameters** (*List*) – List of parameter names that have to be defined.
- **hasToBeSorted** (*Boolean*) – Defines if the TimeSeries has to be sorted or not.
- **hasToBeNormalized** (*Boolean*) – Defines if the TimeSeries has to be normalized or not.

```
_get_parameter_intervals ()
```

Returns the intervals for the methods parameter.

Only parameters with defined intervals can be used for optimization!

**Returns**

Returns a dictionary containing the parameter intervals, using the parameter name as key, while the value has the following format: [minValue, maxValue, minIntervalClosed, maxIntervalClosed]

- **minValue** Minimal value for the parameter

- **maxValue** Maximal value for the parameter
- **minIntervalClosed** `True`, if `minValue` represents a valid value for the parameter. `False` otherwise.
- **maxIntervalClosed** `True`, if `maxValue` represents a valid value for the parameter. `False` otherwise.

**Return type** Dictionary

**`_get_value_error_message_for_invalid_parameter`** (*parameter*, *value*)  
Returns the `ValueError` message for the given parameter.

**Parameters**

- **parameter** (*String*) – Name of the parameter the message has to be created for.
- **value** (*Numeric*) – Value outside the parameters interval.

**Returns** Returns a string containing the message.

**Return type** String

**`_in_valid_interval`** (*parameter*, *value*)  
Returns if the parameter is within its valid interval.

**Parameters**

- **parameter** (*String*) – Name of the parameter that has to be checked.
- **value** (*Numeric*) – Value of the parameter.

**Returns** Returns `True` if the value for the given parameter is valid, `False` otherwise.

**Return type** Boolean

**`can_be_executed`** ()  
Returns if the method can already be executed.

**Returns** Returns `True` if all required parameters were already set, `False` otherwise.

**Return type** Boolean

**`execute`** (*timeSeries*)  
Executes the `BaseMethod` on a given `TimeSeries` object.

**Parameters** **timeSeries** (*TimeSeries*) – `TimeSeries` object that fulfills all requirements (normalization, `sortOrder`).

**Returns** Returns a `TimeSeries` object containing the smoothed/forecasted values.

**Return type** `TimeSeries`

**Raise** Raises a `NotImplementedError` if the child class does not overwrite this function.

**`get_interval`** (*parameter*)  
Returns the interval for a given parameter.

**Parameters** **parameter** (*String*) – Name of the parameter.

**Returns**

Returns a list containing with [`minValue`, `maxValue`, `minIntervalClosed`, `maxIntervalClosed`]. If no interval definitions for the given parameter exist, `None` is returned.

- **minValue** Minimal value for the parameter
- **maxValue** Maximal value for the parameter

- **minIntervalClosed** `True`, if `minValue` represents a valid value for the parameter. `False` otherwise.
- **maxIntervalClosed**: `True`, if `maxValue` represents a valid value for the parameter. `False` otherwise.

**Return type** `List`

**get\_parameter** (*name*)

Returns a forecasting parameter.

**Parameters** **name** (*String*) – Name of the parameter.

**Returns** Returns the value stored in parameter.

**Return type** `Numeric`

**Raise** Raises a `KeyError` if the parameter is not defined.

**get\_required\_parameters** ()

Returns a list with the names of all required parameters.

**Returns** Returns a list with the names of all required parameters.

**Return type** `List`

**has\_to\_be\_normalized** ()

Returns if the `TimeSeries` has to be normalized or not.

**Returns** Returns `True` if the `TimeSeries` has to be normalized, `False` otherwise.

**Return type** `Boolean`

**has\_to\_be\_sorted** ()

Returns if the `TimeSeries` has to be sorted or not.

**Returns** Returns `True` if the `TimeSeries` has to be sorted, `False` otherwise.

**Return type** `Boolean`

**set\_parameter** (*name*, *value*)

Sets a parameter for the `BaseMethod`.

**Parameters**

- **name** (*String*) – Name of the parameter that has to be checked.
- **value** (*Numeric*) – Value of the parameter.

**\_interval\_definitions** = {`False`: ['(', ')'], `True`: ['[', ']']}

```
class pystac.methods.basemethod.BaseForecastingMethod (requiredParameters=[],
                                                         valuesToForecast=1,
                                                         hasToBeSorted=True,
                                                         hasToBeNormalized=True)
```

Bases: `pystac.methods.basemethod.BaseMethod`

Basemethod for all forecasting methods.

**\_\_init\_\_** (*requiredParameters*=[], *valuesToForecast*=1, *hasToBeSorted*=True, *hasToBeNormalized*=True)

Initializes the `BaseForecastingMethod`.

**Parameters**

- **requiredParameters** (*List*) – List of parameter names that have to be defined.

- **valuesToForecast** (*Integer*) – Number of entries that will be forecasted. This can be changed by using `forecast_until()`.
- **hasToBeSorted** (*Boolean*) – Defines if the TimeSeries has to be sorted or not.
- **hasToBeNormalized** (*Boolean*) – Defines if the TimeSeries has to be normalized or not.

**Raise** Raises a `ValueError` when `valuesToForecast` is smaller than zero.

**`_calculate_values_to_forecast`** (*timeSeries*)

Calculates the number of values, that need to be forecasted to match the goal set in `forecast_until`.

This sets the parameter “`valuesToForecast`” and should be called at the beginning of the `BaseMethod.execute()` implementation.

**Parameters** *timeSeries* (*TimeSeries*) – Should be a sorted and normalized TimeSeries instance.

**Raise** Raises a `ValueError` if the TimeSeries is either not normalized or sorted.

**`forecast_until`** (*timestamp, format=None*)

Sets the forecasting goal (timestamp wise).

This function enables the automatic determination of `valuesToForecast`.

**Parameters**

- **timestamp** – timestamp containing the end date of the forecast.
- **format** (*String*) – Format of the timestamp. This is used to convert the timestamp from UNIX epochs, if necessary. For valid examples take a look into the `time.strptime()` documentation.

**`get_optimizable_parameters`** ()

Returns a list with optimizable parameters.

All required parameters of a forecasting method with defined intervals can be used for optimization.

**Returns** Returns a list with optimizable parameter names.

**Return type** List

**Todo** Should we return all parameter names from the `self._parameterIntervals` instead?

**`set_parameter`** (*name, value*)

Sets a parameter for the `BaseForecastingMethod`.

**Parameters**

- **name** (*String*) – Name of the parameter.
- **value** (*Numeric*) – Value of the parameter.

## 4.4.2 Smoothing Methods

**class** `pystac.methods.simplemovingaverage.SimpleMovingAverage` (*windowSize=5*)

Bases: `pystac.methods.basemethod.BaseMethod`

Implements the simple moving average.

The SMA algorithm will calculate the average value at time `t` based on the datapoints between `[t - floor(windowSize / 2), t + floor(windowSize / 2)]`.

**Explanation:** [http://en.wikipedia.org/wiki/Moving\\_average](http://en.wikipedia.org/wiki/Moving_average)

**`__init__`** (*windowSize=5*)

Initializes the `SimpleMovingAverage`.

**Parameters** **windowSize** (*Integer*) – Size of the SimpleMovingAverages window.

**Raise** Raises a `ValueError` if windowSize is an even or not larger than zero.

**`_get_parameter_intervals()`**

Returns the intervals for the methods parameter.

Only parameters with defined intervals can be used for optimization!

**Returns**

Returns a dictionary containing the parameter intervals, using the parameter name as key, while the value has the following format: [minValue, maxValue, minIntervalClosed, maxIntervalClosed]

- **minValue** Minimal value for the parameter
- **maxValue** Maximal value for the parameter
- **minIntervalClosed** `True`, if minValue represents a valid value for the parameter. `False` otherwise.
- **maxIntervalClosed** `True`, if maxValue represents a valid value for the parameter. `False` otherwise.

**Return type** Dictionary

**`execute(timeSeries)`**

Creates a new TimeSeries containing the SMA values for the predefined windowSize.

**Parameters** **timeSeries** (*TimeSeries*) – The TimeSeries used to calculate the simple moving average values.

**Returns** TimeSeries object containing the smooth moving average.

**Return type** TimeSeries

**Raise** Raises a `ValueError` if the defined windowSize is larger than the number of elements in timeSeries

**Note** This implementation aims to support independent for loop execution.

### 4.4.3 Forecasting Methods

**`class pycast.methods.exponentialsmoothing.ExponentialSmoothing`** (*smoothingFactor=0.1, valuesToForecast=1*)

Bases: `pycast.methods.basemethod.BaseForecastingMethod`

Implements an exponential smoothing algorithm.

**Explanation:** <http://www.youtube.com/watch?v=J4iODLa9hYw>

**`__init__`** (*smoothingFactor=0.1, valuesToForecast=1*)

Initializes the ExponentialSmoothing.

**Parameters**

- **smoothingFactor** (*Float*) – Defines the alpha for the ExponentialSmoothing. Valid values are in (0.0, 1.0).
- **valuesToForecast** (*Integer*) – Number of values that should be forecasted.

**Raise** Raises a `ValueError` when smoothingFactor has an invalid value.

**`_get_parameter_intervals()`**

Returns the intervals for the methods parameter.

Only parameters with defined intervals can be used for optimization!

#### Returns

Returns a dictionary containing the parameter intervals, using the parameter name as key, while the value has the following format: [minValue, max\_value, minIntervalClosed, maxIntervalClosed]

- **minValue** Minimal value for the parameter
- **max\_value** Maximal value for the parameter
- **minIntervalClosed** `True`, if min\_value represents a valid value for the parameter. `False` otherwise.
- **maxIntervalClosed** `True`, if max\_value represents a valid value for the parameter. `False` otherwise.

**Return type** Dictionary

**`execute(timeSeries)`**

Creates a new TimeSeries containing the smoothed and forecasted values.

**Returns** TimeSeries object containing the smoothed TimeSeries, including the forecasted values.

**Return type** TimeSeries

**Note** The first normalized value is chosen as the starting point.

**class** `pystat.methods.exponentialsmoothing.HoltMethod` (*smoothingFactor=0.1, trendSmoothingFactor=0.5, valuesToForecast=1*)

Bases: `pystat.methods.basemethod.BaseForecastingMethod`

Implements the Holt algorithm.

**Explanation:** [http://en.wikipedia.org/wiki/Exponential\\_smoothing#Double\\_exponential\\_smoothing](http://en.wikipedia.org/wiki/Exponential_smoothing#Double_exponential_smoothing)

**`__init__`** (*smoothingFactor=0.1, trendSmoothingFactor=0.5, valuesToForecast=1*)

Initializes the HoltMethod.

#### Parameters

- **smoothingFactor** (*Float*) – Defines the alpha for the ExponentialSmoothing. Valid values are in (0.0, 1.0).
- **trendSmoothingFactor** (*Float*) – Defines the beta for the HoltMethod. Valid values are in (0.0, 1.0).
- **valuesToForecast** (*Integer*) – Defines the number of forecasted values that will be part of the result.

**Raise** Raises a `ValueError` when smoothingFactor or trendSmoothingFactor has an invalid value.

**`_get_parameter_intervals()`**

Returns the intervals for the methods parameter.

Only parameters with defined intervals can be used for optimization!



### Returns

Returns a dictionary containing the parameter intervals, using the parameter name as key, while the value has the following format: [minValue, maxValue, minIntervalClosed, maxIntervalClosed]

- **minValue** Minimal value for the parameter
- **maxValue** Maximal value for the parameter
- **minIntervalClosed** `True`, if `minValue` represents a valid value for the parameter. `False` otherwise.
- **maxIntervalClosed** `True`, if `maxValue` represents a valid value for the parameter. `False` otherwise.

**Return type** Dictionary

**execute** (*timeSeries*)

Creates a new TimeSeries containing the smoothed values.

**Returns** TimeSeries object containing the smoothed TimeSeries, including the forecasted values.

**Return type** TimeSeries

**Note** The first normalized value is chosen as the starting point.

```
class pystac.methods.exponentialsmoothing.HoltWintersMethod (smoothingFactor=0.1,
                                                            trendSmoothingFac-
                                                            tor=0.5, seasonS-
                                                            moothingFactor=0.5,
                                                            seasonLength=0, val-
                                                            uesToForecast=0)
```

Bases: `pystac.methods.basemethod.BaseForecastingMethod`

Implements the Holt-Winters algorithm.

**Explanation:** [http://en.wikipedia.org/wiki/Exponential\\_smoothing#Triple\\_exponential\\_smoothing](http://en.wikipedia.org/wiki/Exponential_smoothing#Triple_exponential_smoothing)

```
__init__ (smoothingFactor=0.1, trendSmoothingFactor=0.5, seasonSmoothingFactor=0.5, season-
          Length=0, valuesToForecast=0)
```

Initializes the HoltWintersMethod.

**@param smoothingFactor** Defines the alpha for the Holt-Winters algorithm. Valid values are (0.0, 1.0).

**@param trendSmoothingFactor** Defines the beta for the Holt-Winters algorithm.. Valid values are (0.0, 1.0).

**@param seasonSmoothingFactor** Defines the gamma for the Holt-Winters algorithm. Valid values are (0.0, 1.0).

**@param seasonLength** The expected length for the seasons. Please use a good estimate here! **@param valuesToForecast** Defines the number of forecasted values that will

be part of the result.

```
_get_parameter_intervals ()
```

Returns the intervals for the methods parameter.

Only parameters with defined intervals can be used for optimization!

**Returns**

Returns a dictionary containing the parameter intervals, using the parameter name as key, while the value has the following format: [minValue, maxValue, minIntervalClosed, maxIntervalClosed]

- **minValue** Minimal value for the parameter
- **maxValue** Maximal value for the parameter
- **minIntervalClosed** True, if minValue represents a valid value for the parameter. False otherwise.
- **maxIntervalClosed** True, if maxValue represents a valid value for the parameter. False otherwise.

**Return type** Dictionary

**computeA** (*j, timeSeries*)

Calculates A<sub>j</sub>. A<sub>j</sub> is the average value of x in the jth cycle of your data

@return A<sub>j</sub>

**execute** (*timeSeries*)

Creates a new TimeSeries containing the smoothed values.

@return TimeSeries object containing the exponentially smoothed TimeSeries, including the forecasted values.

@todo Double check if it is correct not to add the first original value to the result. @todo Currently the first normalized value is simply chosen as the starting point.

@throw Throws a NotImplementedError if the child class does not overwrite this function.

**initSeasonFactors** (*timeSeries*)

Computes the initial season smoothing factors.

@return a list of season vectors of length "seasonLength"

**initialTrendSmoothingFactors** (*timeSeries*)

Calculate the initial Trend smoothing Factor b0 according to:  
[http://en.wikipedia.org/wiki/Exponential\\_smoothing#Triple\\_exponential\\_smoothing](http://en.wikipedia.org/wiki/Exponential_smoothing#Triple_exponential_smoothing)

@return initial Trend smoothing Factor b0

#### 4.4.4 Custom Methods

Custom smoothing and forecasting methods must either inherit from *pystat.methods.BaseMethod* or *pystat.methods.BaseForecastingMethod* and implement the following functions:

- `__init__(self, *args, **kwargs)`
- `execute(self, timeSeries)`
- `get_parameter_intervals(self)`

#### Code to start with

To implement your custom method, it is recommended to start with the following example:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

#Copyright (c) 2012 Christian Schwarz
```

```
#
#Permission is hereby granted, free of charge, to any person obtaining
#a copy of this software and associated documentation files (the
#"Software"), to deal in the Software without restriction, including
#without limitation the rights to use, copy, modify, merge, publish,
#distribute, sublicense, and/or sell copies of the Software, and to
#permit persons to whom the Software is furnished to do so, subject to
#the following conditions:
#
#The above copyright notice and this permission notice shall be
#included in all copies or substantial portions of the Software.
#
#THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
#EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
#MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
#NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
#LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
#OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
#WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

from pystac.methods import BaseMethod
from pystac.common.timeseries import TimeSeries

class CustomSmoothingMethod(BaseMethod):
    ## Alternative:
    ### class CustomForecastingMethod(BaseMethod):
    """This is your custom Method"""

    def __init__(self, *args, **kwargs):
        """Initializes the BaseMethod.

        :param List args: Arguments that are required for initialization.
        :param Dictionary hasToBeSorted: Keyword arguments that are required for initialization.
        """
        super(BaseMethod, self).__init__(requiredParameters, hasToBeSorted, hasToBeNormalized)

        ## YOUR CUSTOM CODE HERE

    def _get_parameter_intervals(self):
        """Returns the intervals for the methods parameter.

        Only parameters with defined intervals can be used for optimization!

        :return: Returns a dictionary containing the parameter intervals, using the parameter
            name as key, while the value has the following format:

            [minValue, maxValue, minIntervalClosed, maxIntervalClosed]

            - minValue
              Minimal value for the parameter

            - maxValue
              Maximal value for the parameter

            - minIntervalClosed
              :py:const:`True`, if minValue represents a valid value for the parameter.
              :py:const:`False` otherwise.
```

```

        - maxIntervalClosed:
            :py:const:`True`, if maxInterval represents a valid value for the parameter.
            :py:const:`False` otherwise.

    :rtype:      Dictionary
    """
    parameterIntervals = {}

    ## YOUR METHOD SPECIFIC CODE HERE!

    return parameterIntervals

def execute(self, timeSeries):
    """Executes the BaseMethod on a given TimeSeries object.

    :param TimeSeries timeSeries: TimeSeries object that fullfills all requirements (normalization, s

    :return:      Returns a TimeSeries object containing the smoothed/forecasted values.
    :rtype:      TimeSeries

    :raise:      Raises a :py:exc:`NotImplementedError` if the child class does not overwrite this fun
    """
    ## YOUR METHOD SPECIFIC CODE HERE!

```

## 4.5 Error Measures

### 4.5.1 BaseErrorMeasure

**class** `pystac.errors.baseerrormmeasure.BaseErrorMeasure` (*minimalErrorCalculationPercentage=60*)  
 Bases: `object`

Baseclass for all error measures.

**\_\_init\_\_** (*minimalErrorCalculationPercentage=60*)  
 Initializes the error measure.

**Parameters** *minimalErrorCalculationPercentage* (*Integer*) – The number of entries in an original TimeSeries that have to have corresponding partners in the calculated TimeSeries. Corresponding partners have the same time stamp. Valid values are in [0.0, 100.0].

**Raise** Raises a `ValueError` if *minimalErrorCalculationPercentage* is not in [0.0, 100.0].

**\_calculate** (*startingPercentage, endPercentage*)  
 This is the error calculation function that gets called by `BaseErrorMeasure.get_error()`.

Both parameters will be correct at this time.

**Parameters**

- **startingPercentage** (*Float*) – Defines the start of the interval. This has to be a value in [0.0, 100.0]. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25% of all calculated errors will be ignored.
- **endPercentage** (*Float*) – Defines the end of the interval. This has to be a value in [0.0, 100.0]. It represents the vlaue, after which all error values will be ignored. 90.0 for example means that the last 10% of all local errors will be ignored.

**Returns** Returns a float representing the error.

**Return type** Float

**Raise** Raises a `NotImplementedError` if the child class does not overwrite this method.

**`_get_error_values`** (*startingPercentage*, *endPercentage*)

Gets the defined subset of `self._errorValues`.

Both parameters will be correct at this time.

**Parameters**

- **`startingPercentage`** (*Float*) – Defines the start of the interval. This has to be a value in `[0.0, 100.0]`. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25% of all calculated errors will be ignored.
- **`endPercentage`** (*Float*) – Defines the end of the interval. This has to be a value in `[0.0, 100.0]`. It represents the value, after which all error values will be ignored. 90.0 for example means that the last 10% of all local errors will be ignored.

**Returns** Returns a list with the defined error values.

**Return type** List

**`get_error`** (*startingPercentage=0.0*, *endPercentage=100.0*)

Calculates the error for the given interval (*startingPercentage*, *endPercentage*) between the `TimeSeries` given during `BaseErrorMeasure.initialize()`.

**Parameters**

- **`startingPercentage`** (*Float*) – Defines the start of the interval. This has to be a value in `[0.0, 100.0]`. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25% of all calculated errors will be ignored.
- **`endPercentage`** (*Float*) – Defines the end of the interval. This has to be a value in `[0.0, 100.0]`. It represents the value, after which all error values will be ignored. 90.0 for example means that the last 10% of all local errors will be ignored.

**Returns** Returns a float representing the error.

**Return type** Float

**Raise** Raises a `ValueError` in one of the following cases:

- `startingPercentage` not in `[0.0, 100.0]`
- `endPercentage` not in `[0.0, 100.0]`
- `endPercentage < startingPercentage`

**Raise** Raises a `StandardError` if `BaseErrorMeasure.initialize()` was not successful before.

**`initialize`** (*originalTimeSeries*, *calculatedTimeSeries*)

Initializes the `ErrorMeasure`.

During initialization, all `BaseErrorMeasure.local_errors()` are calculated.

**Parameters**

- **`originalTimeSeries`** (*TimeSeries*) – `TimeSeries` containing the original data.
- **`calculatedTimeSeries`** (*TimeSeries*) – `TimeSeries` containing calculated data. Calculated data is smoothed or forecasted data.

**Returns** Return `True` if the error could be calculated, `False` otherwise based on the `minimalErrorCalculationPercentage`.

**Return type** Boolean

**Raise** Raises a `StandardError` if the error measure is initialized multiple times.

**local\_error** (*originalValue*, *calculatedValue*)

Calculates the error between the two given values.

**Parameters**

- **originalValue** (*Numeric*) – Value of the original data.
- **calculatedValue** (*Numeric*) – Value of the calculated `TimeSeries` that corresponds to `originalValue`.

**Returns** Returns the error measure of the two given values.

**Return type** Numeric

**Raise** Raises a `NotImplementedError` if the child class does not overwrite this method.

## 4.5.2 Mean Squared Error

**class** `pystac.errors.meansquarederror.MeanSquaredError` (*minimalErrorCalculationPercentage=60*)

Bases: `pystac.errors.baseerrormmeasure.BaseErrorMeasure`

Implements the mean squared error measure.

**Explanation:** [http://en.wikipedia.org/wiki/Mean\\_squared\\_error](http://en.wikipedia.org/wiki/Mean_squared_error)

**\_calculate** (*startingPercentage*, *endPercentage*)

This is the error calculation function that gets called by `BaseErrorMeasure.get_error()`.

Both parameters will be correct at this time.

**Parameters**

- **startingPercentage** (*Float*) – Defines the start of the interval. This has to be a value in `[0.0, 100.0]`. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25% of all calculated errors will be ignored.
- **endPercentage** (*Float*) – Defines the end of the interval. This has to be a value in `[0.0, 100.0]`. It represents the value, after which all error values will be ignored. 90.0 for example means that the last 10% of all local errors will be ignored.

**Returns** Returns a float representing the error.

**Return type** Float

**Raise** Raises a `NotImplementedError` if the child class does not overwrite this method.

**local\_error** (*originalValue*, *calculatedValue*)

Calculates the error between the two given values.

**Parameters**

- **originalValue** (*Numeric*) – Value of the original data.
- **calculatedValue** (*Numeric*) – Value of the calculated `TimeSeries` that corresponds to `originalValue`.

**Returns** Returns the error measure of the two given values.

**Return type** Float

### 4.5.3 Mean Absolute Deviation

`class pycast.errors.meanabsolutedeviceerror.MeanAbsoluteDeviationError (minimalErrorCalculationPe`  
 Bases: `pycast.errors.baseerrormeasure.BaseErrorMeasure`

Implements the mean absolute deviation error measure.

`_calculate (startingPercentage, endPercentage)`

This is the error calculation function that gets called by `BaseErrorMeasure.get_error()`.

Both parameters will be correct at this time.

#### Parameters

- **startingPercentage** (*Float*) – Defines the start of the interval. This has to be a value in [0.0, 100.0]. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25% of all calculated errors will be ignored.
- **endPercentage** (*Float*) – Defines the end of the interval. This has to be a value in [0.0, 100.0]. It represents the vlaue, after which all error values will be ignored. 90.0 for example means that the last 10% of all local errors will be ignored.

**Returns** Returns a float representing the error.

**Return type** Float

`local_error (originalValue, calculatedValue)`

Calculates the error between the two given values.

#### Parameters

- **originalValue** (*Numeric*) – Value of the original data.
- **calculatedValue** (*Numeric*) – Value of the calculated TimeSeries that corresponds to originalValue.

**Returns** Returns the error measure of the two given values.

**Return type** Numeric

### 4.5.4 Mean Absolute Percentage Error

`class pycast.errors.meanabsolutepercentageerror.MeanAbsolutePercentageError (minimalErrorCalculation`  
 Bases: `pycast.errors.baseerrormeasure.BaseErrorMeasure`

`_calculate (startingPercentage, endPercentage)`

This is the error calculation function that gets called by `get_error()`.

Both parameters will be correct at this time.

#### Parameters

- **startingPercentage** (*Float*) – Defines the start of the interval. This has to be a float value in [0.0, 100.0]. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25%% of all calculated errors will be ignored.
- **endPercentage** (*Float*) – Defines the end of the interval. This has to be a float value in [0.0, 100.0]. It represents the vlaue, after which all error values will be ignored. 90.0 for example means that the last 10%% of all local errors will be ignored.

**Returns** Returns a float representing the error.

**Return type** Float

**local\_error** (*originalValue, calculatedValue*)

Calculates the error between the two given values.

**Parameters**

- **originalValue** (*Numeric*) – Value of the original data.
- **calculatedValue** (*Numeric*) – Value of the calculated TimeSeries that corresponds to originalValue.

**Returns** Returns the error measure of the two given values.

**Return type** Numeric

## 4.5.5 Geometric Mean Absolute Percentage Error

**class** `pystac.errors.meanabsolutepercentageerror.GeometricMeanAbsolutePercentageError` (*minimalErrorCalc*)

Bases: `pystac.errors.meanabsolutepercentageerror.MeanAbsolutePercentageError`

Calculates the geometric MAPE.

**\_calculate** (*startingPercentage, endPercentage*)

This is the error calculation function that gets called by `get_error()`.

Both parameters will be correct at this time.

**Parameters**

- **startingPercentage** (*Float*) – Defines the start of the interval. This has to be a float value in [0.0, 100.0]. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25%% of all calculated errors will be ignored.
- **endPercentage** (*Float*) – Defines the end of the interval. This has to be a float value in [0.0, 100.0]. It represents the value, after which all error values will be ignored. 90.0 for example means that the last 10%% of all local errors will be ignored.

**Returns** Returns a float representing the error.

**Return type** Float

## 4.5.6 Median Absolute Percentage Error

**class** `pystac.errors.medianabsolutepercentageerror.MedianAbsolutePercentageError` (*minimalErrorCalc*)

Bases: `pystac.errors.meanabsolutepercentageerror.MeanAbsolutePercentageError`

Represents the median absolute percentage error.

**\_calculate** (*startingPercentage, endPercentage*)

This is the error calculation function that gets called by `get_error()`.

Both parameters will be correct at this time.

**Parameters**

- **startingPercentage** (*Float*) – Defines the start of the interval. This has to be a float value in [0.0, 100.0]. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25%% of all calculated errors will be ignored.
- **endPercentage** (*Float*) – Defines the end of the interval. This has to be a float value in [0.0, 100.0]. It represents the value, after which all error values will be ignored. 90.0 for example means that the last 10%% of all local errors will be ignored.



**Returns** Returns a float representing the error.

**Return type** Float

## 4.5.7 Symmetric Mean Absolute Percentage Error

**class** `pycast.errors.symmetricmeanabsolutepercentageerror.SymmetricMeanAbsolutePercentageError`

Bases: `pycast.errors.baseerrormeasure.BaseErrorMeasure`

Implements the symmetric mean absolute percentage error with a boarder of 200%.

**Explanation:** <http://monashforecasting.com/index.php?title=SMAPE> (Formula (3))

If the calculated value and the original value are equal, the error is 0.

**\_calculate** (*startingPercentage, endPercentage*)

This is the error calculation function that gets called by `BaseErrorMeasure.get_error()`.

Both parameters will be correct at this time.

### Parameters

- **startingPercentage** (*Float*) – Defines the start of the interval. This has to be a value in [0.0, 100.0]. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25% of all calculated errors will be ignored.
- **endPercentage** (*Float*) – Defines the end of the interval. This has to be a value in [0.0, 100.0]. It represents the vlaue, after which all error values will be ignored. 90.0 for example means that the last 10% of all local errors will be ignored.

**Returns** Returns a float representing the error.

**Return type** Float

**local\_error** (*originalValue, calculatedValue*)

Calculates the error between the two given values.

### Parameters

- **originalValue** (*Numeric*) – Value of the original data.
- **calculatedValue** (*Numeric*) – Value of the calculated TimeSeries that corresponds to originalValue.

**Returns** Returns the error measure of the two given values.

**Return type** Numeric

## 4.5.8 Custom Error Measures

Custom error measures must inherit from `pycast.errors.BaseErrorMeasure` and implement the following functions:

- `_calculate(self, startingPercentage, endPercentage)`
- `local_error(self, originalValue, calculatedValue)`

### Code to start with

To implement your custom error measrue, it is recommended to start with the following example:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

#Copyright (c) 2012 Christian Schwarz
#
#Permission is hereby granted, free of charge, to any person obtaining
#a copy of this software and associated documentation files (the
#"Software"), to deal in the Software without restriction, including
#without limitation the rights to use, copy, modify, merge, publish,
#distribute, sublicense, and/or sell copies of the Software, and to
#permit persons to whom the Software is furnished to do so, subject to
#the following conditions:
#
#The above copyright notice and this permission notice shall be
#included in all copies or substantial portions of the Software.
#
#THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
#EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
#MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
#NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
#LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
#OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
#WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

from pystac.errors import BaseErrorMeasure

class CustomErrorMeasure(BaseErrorMeasure):
    """Implements your custom error measure.

    Please provide some explanation here!
    """

    def _calculate(self, startingPercentage, endPercentage):
        """This is the error calculation function that gets called by :py:meth:`BaseErrorMeasure.get_

        Both parameters will be correct at this time.

        :param Float startingPercentage: Defines the start of the interval. This has to be a value in [0, 100]
        It represents the value, where the error calculation should be started.
        25.0 for example means that the first 25% of all calculated errors will be ignored.
        :param Float endPercentage: Defines the end of the interval. This has to be a value in [0, 100]
        It represents the vlaue, after which all error values will be ignored. 90.0 for example means that
        the last 10% of all local errors will be ignored.

        :return: Returns a float representing the error.
        :rtype: Float

        :raise: Raises a :py:exc:`NotImplementedError` if the child class does not overwrite this
        """
        ## get the defined subset of error values
        errorValues = self._get_error_values(startingPercentage, endPercentage)

        ## Implement the error calculation here!

    def local_error(self, originalValue, calculatedValue):
        """Calculates the error between the two given values.

        :param Numeric originalValue: Value of the original data.
```

```

:param Numeric calculatedValue:    Value of the calculated TimeSeries that
    corresponds to originalValue.

:return:    Returns the error measure of the two given values.
:rtype:    Numeric

:raise:    Raises a :py:exc:`NotImplementedError` if the child class does not overwrite this
    """
    ## Implement the local error calculation here!

```

## 4.6 Optimization Methods

```

class pystac.optimization.baseoptimizationmethod.BaseOptimizationMethod(errorMeasureClass,
    er-
    rorMea-
    sureIni-
    tial-
    iza-
    tion-
    Pa-
    rame-
    ters={},
    precision=-
    1)

```

Bases: object

Baseclass for all optimization methods.

**\_\_init\_\_** (errorMeasureClass, errorMeasureInitializationParameters={}, precision=-1)  
 Initializes the optimization method.

### Parameters

- **errorMeasureClass** (*BaseErrorMeasure*) – Error measure class from `pystac.errors`
- **errorMeasureInitializationParameters** (*Dictionary*) – Parameters used to initialize the errorMeasureClass. This dictionary will be passed to the errorMeasureClass as **\*\*kwargs**.
- **precision** (*Integer*) – Defines the accuracy for parameter tuning in  $10^{\text{precision}}$ . This parameter has to be an integer in  $[-7, 0]$ .

**Raise** Raises a `TypeError` if errorMeasureClass is not a valid class. Valid classes are derived from `pystac.errors.BaseErrorMeasure`.

**Raise** Raises a `ValueError` if precision is not in  $[-7, 0]$ .

**optimize** (timeSeries, forecastingMethods=[])  
 Runs the optimization on the given TimeSeries.

### Parameters

- **timeSeries** (*TimeSeries*) – TimeSeries instance that requires an optimized forecast.
- **forecastingMethods** (*List*) – List of forecastingMethods that will be used for optimization.

**Returns** Returns the optimized forecasting method with the smallest error.

**Return type** (`BaseForecastingMethod`, `Dictionary`)

**Raise** Raises a `ValueError` if no forecastingMethods is empty.

**class** `pystac.optimization.gridsearch.GridSearch` (*errorMeasureClass*, *errorMeasureInitializationParameters*=`{}`, *precision*=`-1`)

Bases: `pystac.optimization.baseoptimizationmethod.BaseOptimizationMethod`

Implements the grid search method for parameter optimization.

GridSearch is the brute force method.

**`_generate_next_parameter_value`** (*parameter*, *forecastingMethod*)

Generator for a specific parameter of the given forecasting method.

#### Parameters

- **parameter** (*String*) – Name of the parameter the generator is used for.
- **forecastingMethod** (*BaseForecastingMethod*) – Instance of a ForecastingMethod.

**Returns** Creates a generator used to iterate over possible parameters.

**Return type** Generator Function

**`optimization_loop`** (*timeSeries*, *forecastingMethod*, *remainingParameters*, *currentParameterValues*=`{}`)

The optimization loop.

This function is called recursively, until all parameter values were evaluated.

#### Parameters

- **timeSeries** (*TimeSeries*) – TimeSeries instance that requires an optimized forecast.
- **forecastingMethod** (*BaseForecastingMethod*) – ForecastingMethod that is used to optimize the parameters.
- **remainingParameters** (*list*) – List containing all parameters with their corresponding values that still need to be evaluated. When this list is empty, the most inner optimization loop is reached.
- **currentParameterValues** (*Dictionary*) – The currently evaluated forecast parameter combination.

**Returns** Returns a list containing a BaseErrorMeasure instance as defined in BaseOptimizationMethod.\_\_init\_\_() and the forecastingMethods parameter.

**Return type** List

**`optimize`** (*timeSeries*, *forecastingMethods*=`[]`, *startingPercentage*=`0.0`, *endPercentage*=`100.0`)

Runs the optimization of the given TimeSeries.

#### Parameters

- **timeSeries** (*TimeSeries*) – TimeSeries instance that requires an optimized forecast.
- **forecastingMethods** (*List*) – List of forecastingMethods that will be used for optimization.
- **startingPercentage** (*Float*) – Defines the start of the interval. This has to be a value in [0.0, 100.0]. It represents the value, where the error calculation should be started. 25.0 for example means that the first 25% of all calculated errors will be ignored.
- **endPercentage** (*Float*) – Defines the end of the interval. This has to be a value in [0.0, 100.0]. It represents the value, after which all error values will be ignored. 90.0 for example means that the last 10% of all local errors will be ignored.

**Returns** Returns the optimized forecasting method, the corresponding error measure and the forecasting methods parameters.

**Return type** [BaseForecastingMethod, BaseErrorMeasure, Dictionary]

**Raise** Raises a `ValueError` if no forecastingMethods is empty.

**optimize\_forecasting\_method** (*timeSeries*, *forecastingMethod*)

Optimizes the parameters for the given timeSeries and forecastingMethod.

**Parameters**

- **timeSeries** (*TimeSeries*) – TimeSeries instance, containing the original data.
- **forecastingMethod** (*BaseForecastingMethod*) – ForecastingMethod that is used to optimize the parameters.

**Returns** Returns a tuple containing only the smallest BaseErrorMeasure instance as defined in `BaseOptimizationMethod.__init__()` and the forecastingMethods parameter.

**Return type** Tuple

## 4.7 pycast conventions

pycast uses several conventions to make your life easier. While Python does not enforce those, you are strongly recommended to follow our guidelines ;).

### 4.7.1 Naming Conventions

When using pycast variables, objects, methods and functions, the following naming conventions should help you to find what you

- variables, functions, methods, classes and modules starting with an underscore (`_`) should be seen as private, inheritable attributes of any entity
- functions and methods use underscores (`_`) to delimit words while variables and classes use CamelCase

### 4.7.2 Documentation Conventions

pycast variables, objects, methods and functions are documented using Sphinx. When writing documentation, the following docstring can be used as starting point.

```
1  """Some short info here.
2
3  Some more detailed info here. If you need to write a list, use the following:
4
5      - List item One
6      - List item Two
7
8  :param Type parameterNameFromSignature: This is the documentation for the first
9      parameter. It has some additional Type information if a specific Type is expected.
10     Default data types are: Integer, Float, Numeric, String, List, Dictionary, Tuple,
11     ClassNames, ...
12  :param Type secondParameterNameFromSignature: This is the documentation for
13     the second parameter. Do not use an empty line between your parameters!
14
15  :return: If the function has a valid return value (other than :py:const:`None`),
16     this should be used, including the following:
17  :rtype: The return type.
18
```

```
19 :raise:    A short message, when a specific :py:exc:`Exception` is raised.
20
21 :warning:   A warning if anything can be potentially dangerous.
22
23 :note:      An internal explanation for implementation details if not mentioned earlier.
24 :todo:      A reminder for internal development discussions.
25 """
```

## USEFUL LINKS

- [github Project Page](#)
- *search*





# PYTHON MODULE INDEX

## p

`pycast.common.helper`, 9

`pycast.common.profileme`, 9



# INDEX

## Symbols

- `_ProfileDecorator` (class in `pycast.common.profileme`), 9
- `__add__()` (`pycast.common.timeseries.TimeSeries` method), 10
- `__call__()` (`pycast.common.profileme._ProfileDecorator` method), 9
- `__copy__()` (`pycast.common.timeseries.TimeSeries` method), 11
- `__eq__()` (`pycast.common.timeseries.TimeSeries` method), 11
- `__getitem__()` (`pycast.common.timeseries.TimeSeries` method), 11
- `__init__()` (`pycast.common.profileme._ProfileDecorator` method), 9
- `__init__()` (`pycast.common.timeseries.TimeSeries` method), 11
- `__init__()` (`pycast.errors.baseerrormeasure.BaseErrorMeasure` method), 24
- `__init__()` (`pycast.methods.basemethod.BaseForecastingMethod` method), 17
- `__init__()` (`pycast.methods.basemethod.BaseMethod` method), 15
- `__init__()` (`pycast.methods.exponentialsMOOTHING.ExponentialSmoothing` method), 19
- `__init__()` (`pycast.methods.exponentialsMOOTHING.HoltMethod` method), 20
- `__init__()` (`pycast.methods.exponentialsMOOTHING.HoltWintersMethod` method), 21
- `__init__()` (`pycast.methods.simplemovingaverage.SimpleMovingAverage` method), 18
- `__init__()` (`pycast.optimization.baseoptimizationmethod.BaseOptimizationMethod` method), 31
- `__iter__()` (`pycast.common.timeseries.TimeSeries` method), 11
- `__len__()` (`pycast.common.timeseries.TimeSeries` method), 11
- `__setitem__()` (`pycast.common.timeseries.TimeSeries` method), 12
- `__str__()` (`pycast.common.timeseries.TimeSeries` method), 12
- `_calculate()` (`pycast.errors.baseerrormeasure.BaseErrorMeasure` method), 24
- `_calculate()` (`pycast.errors.meanabsolutedeviationerror.MeanAbsoluteDeviationError` method), 27
- `_calculate()` (`pycast.errors.meanabsolutepercentageerror.GeometricMeanAbsolutePercentageError` method), 28
- `_calculate()` (`pycast.errors.meanabsolutepercentageerror.MeanAbsolutePercentageError` method), 27
- `_calculate()` (`pycast.errors.meansquarederror.MeanSquaredError` method), 26
- `_calculate()` (`pycast.errors.medianabsolutepercentageerror.MedianAbsolutePercentageError` method), 28
- `_calculate()` (`pycast.errors.symmetricmeanabsolutepercentageerror.SymmetricMeanAbsolutePercentageError` method), 29
- `_calculate_values_to_forecast()` (`pycast.methods.basemethod.BaseForecastingMethod` method), 18
- `_generate_next_parameter_value()` (`pycast.optimization.gridsearch.GridSearch` method), 32
- `_get_error_values()` (`pycast.errors.baseerrormeasure.BaseErrorMeasure` method), 25
- `_get_parameter_intervals()` (`pycast.methods.basemethod.BaseMethod` method), 15
- `_get_parameter_intervals()` (`pycast.methods.exponentialsMOOTHING.ExponentialSmoothing` method), 19
- `_get_parameter_intervals()` (`pycast.methods.exponentialsMOOTHING.HoltMethod` method), 20
- `_get_parameter_intervals()` (`pycast.methods.exponentialsMOOTHING.HoltWintersMethod` method), 21
- `_get_parameter_intervals()` (`pycast.methods.simplemovingaverage.SimpleMovingAverage` method), 19
- `_get_value_error_message_for_invalid_parameter()` (`pycast.methods.basemethod.BaseMethod` method), 16

`_in_valid_interval()`  
     `cast.methods.basemethod.BaseMethod`  
     method), 16

`_interval_definitions`  
     `cast.methods.basemethod.BaseMethod`  
     tribute), 17

## A

`add_entry()` (pycast.common.timeseries.TimeSeries  
     method), 12

`apply()` (pycast.common.timeseries.TimeSeries  
     method), 12

## B

`BaseErrorMeasure` (class in py-  
     `cast.errors.baseerrormeasure`), 24

`BaseForecastingMethod` (class in py-  
     `cast.methods.basemethod`), 17

`BaseMethod` (class in `pycast.methods.basemethod`), 15

`BaseOptimizationMethod` (class in py-  
     `cast.optimization.baseoptimizationmethod`),  
     31

## C

`can_be_executed()`  
     `cast.methods.basemethod.BaseMethod`  
     method), 16

`check_normalization()` (py-  
     `cast.common.timeseries.TimeSeries` method),  
     12

`computeA()` (pycast.methods.exponentialsMOOTHING.HoltWintersMethod  
     method), 22

`convert_epoch_to_timestamp()` (py-  
     `cast.common.timeseries.TimeSeries` class  
     method), 12

`convert_timestamp_to_epoch()` (py-  
     `cast.common.timeseries.TimeSeries` class  
     method), 13

## E

`execute()` (pycast.methods.basemethod.BaseMethod  
     method), 16

`execute()` (pycast.methods.exponentialsMOOTHING.ExponentialSmoothing  
     method), 20

`execute()` (pycast.methods.exponentialsMOOTHING.HoltMethod  
     method), 21

`execute()` (pycast.methods.exponentialsMOOTHING.HoltWintersMethod  
     method), 22

`execute()` (pycast.methods.simplemovingaverage.SimpleMovingAverage  
     method), 19

`ExponentialSmoothing` (class in py-  
     `cast.methods.exponentialsMOOTHING`), 19

## F

`forecast_until()` (pycast.methods.basemethod.BaseForecastingMethod  
     method), 18

`from_json()` (pycast.common.timeseries.TimeSeries class  
     method), 13

`from_twodim_list()` (py-  
     `cast.common.timeseries.TimeSeries` class  
     method), 13

## G

`GeometricMeanAbsolutePercentageError` (class in py-  
     `cast.errors.meanabsolutepercentageerror`), 28

`get_error()` (pycast.errors.baseerrormeasure.BaseErrorMeasure  
     method), 25

`get_interval()` (pycast.methods.basemethod.BaseMethod  
     method), 16

`get_optimizable_parameters()` (py-  
     `cast.methods.basemethod.BaseForecastingMethod`  
     method), 18

`get_parameter()` (pycast.methods.basemethod.BaseMethod  
     method), 17

`get_required_parameters()` (py-  
     `cast.methods.basemethod.BaseMethod`  
     method), 17

`GridSearch` (class in `pycast.optimization.gridsearch`), 31

## H

`has_to_be_normalized()` (py-  
     `cast.methods.basemethod.BaseMethod`  
     method), 17

`is_sorted()` (py-  
     `cast.methods.basemethod.BaseMethod`  
     method), 17

`HoltMethod` (class in py-  
     `cast.methods.exponentialsMOOTHING`), 20

`HoltWintersMethod` (class in py-  
     `cast.methods.exponentialsMOOTHING`), 21

## I

`initialize()` (pycast.errors.baseerrormeasure.BaseErrorMeasure  
     method), 25

`initialize_from_sql_cursor()` (py-  
     `cast.common.timeseries.TimeSeries` method),  
     13

`initialTrendSmoothingFactors()` (py-  
     `cast.methods.exponentialsMOOTHING.HoltWintersMethod`  
     method), 22

`initSeasonFactors()` (py-  
     `cast.methods.exponentialsMOOTHING.HoltWintersMethod`  
     method), 22

`is_normalized()` (pycast.common.timeseries.TimeSeries  
     method), 14

`is_sorted()` (pycast.common.timeseries.TimeSeries  
     method), 14

## L

linear\_interpolation() (in module pycast.common.helper), 9

local\_error() (pycast.errors.baseerrormeasure.BaseErrorMeasure method), 26

local\_error() (pycast.errors.meanabsolutedeviationerror.MeanAbsoluteDeviationError method), 27

local\_error() (pycast.errors.meanabsolutepercentageerror.MeanAbsolutePercentageError method), 27

local\_error() (pycast.errors.meansquarederror.MeanSquaredError method), 26

local\_error() (pycast.errors.symmetricmeanabsolutepercentageerror.SymmetricMeanAbsolutePercentageError method), 29

set\_timeformat() (pycast.common.timeseries.TimeSeries method), 14

SimpleMovingAverage (class in pycast.methods.simplerovingaverage), 18

sort\_timeseries() (pycast.common.timeseries.TimeSeries method), 14

sorted\_timeseries() (pycast.common.timeseries.TimeSeries method), 14

SymmetricMeanAbsolutePercentageError (class in pycast.errors.symmetricmeanabsolutepercentageerror), 29

## M

MeanAbsoluteDeviationError (class in pycast.errors.meanabsolutedeviationerror), 27

MeanAbsolutePercentageError (class in pycast.errors.meanabsolutepercentageerror), 27

MeanSquaredError (class in pycast.errors.meansquarederror), 26

MedianAbsolutePercentageError (class in pycast.errors.medianabsolutepercentageerror), 28

TimeSeries (class in pycast.common.timeseries), 10

to\_gnuplot\_datafile() (pycast.common.timeseries.TimeSeries method), 15

to\_json() (pycast.common.timeseries.TimeSeries method), 15

to\_twodim\_list() (pycast.common.timeseries.TimeSeries method), 15

## N

normalize() (pycast.common.timeseries.TimeSeries method), 14

## O

optimization\_loop() (pycast.optimization.gridsearch.GridSearch method), 32

optimize() (pycast.optimization.baseoptimizationmethod.BaseOptimizationMethod method), 31

optimize() (pycast.optimization.gridsearch.GridSearch method), 32

optimize\_forecasting\_method() (pycast.optimization.gridsearch.GridSearch method), 33

## P

profileMe (in module pycast.common.profileme), 9

pycast.common.helper (module), 9

pycast.common.profileme (module), 9

## S

set\_parameter() (pycast.methods.basemethod.BaseForecastingMethod method), 18

set\_parameter() (pycast.methods.basemethod.BaseMethod method), 17