pycast Documentation

Release v0.0.9-prealpha

Christian Schwarz

November 20, 2012

CONTENTS

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`

CHAPTER

ONE

REQUIREMENTS

pycast

- nose >= 1.2.1
- coverage >= 3.5.3

Documentation

- sphinx >= 1.1.3

Examples

- itty >= 0.8.1

CHAPTER

TWO

LICENSE

CHAPTER

THREE

CONTRIBUTORS

Christian Schwarz, Felix Leupold

CHAPTER

FOUR

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 pycast.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 `pycast.common.TimeSeries.normalize()` are stored in `pycast.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 `pycast.common.TimeSeries.normalize()` are stored in `pycast.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 `pycast.common.TimeSeries.normalize()` are stored in `pycast.common.timeseries.InterpolationMethods`.

Valid values for interpolation methods are:

- "linear": Use linear interpolation to calculate the missing values

4.3 TimeSeries

class `pycast.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

__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 realy 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 `pycast.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.

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 assumtions 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 hanve 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 occurence inline.

Parameters **ascending** (*Boolean*) – Determines if the TimeSeries will be ordered ascending or decending. If this is set to decending 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 assumtion 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 decending.

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 Base Methods

class pycast.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 parameternames 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 hast 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_prarameter(*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 hte 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` it 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 where already set, False otherwise.

Return type Boolean

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

Parameters **timeSeries** (*TimeSeries*) – TimeSeries object that fullfills 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 pycast.methods.basemethod.**BaseForecastingMethod**(*requiredParameters=[]*, *valuesToForecast=1*, *hasToBeSorted=True*, *hasToBeNormalized=True*)
Bases: pycast.methods.basemethod.BaseMethod

Basemethod for all forecasting methods.

__init__(*requiredParameters=[]*, *valuesToForecast=1*, *hasToBeSorted=True*, *hasToBeNormalized=True*)
Initializes the BaseForecastingMethod.

Parameters

- **requiredParameters** (*List*) – List of parameternames 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.

4.5 Smoothing Methods

class pycast.methods.simplemovingaverage.**SimpleMovingAverage** (*windowsize=5*)
Bases: pycast.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

__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.

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

Note This implementation aims to support independent for loop execution.

4.6 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 hast 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 and forcasted 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 pycast.methods.exponentialsmoothing.**HoltMethod**(*smoothingFactor=0.1*, *trendSmoothingFactor=0.5*, *valuesToForecast=1*)

Bases: pycast.methods.basemethod.BaseForecastingMethod

Implements the Holt algorithm.

Explanation: 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 hast 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.

4.7 Error Measures

4.7.1 BaseErrorMeasure

class pycast.errors.baseerrormeasure.**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 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 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 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 `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 successfull 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.

4.7.2 Mean Squared Error

class `pycast.errors.meansquarederror.`**MeanSquaredError**(*minimalErrorCalculationPercentage=60*)
Bases: `pycast.errors.baseerrormeasure.BaseErrorMeasure`

Implements the mean squared error measure.

Explanation: 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 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.

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

4.7.3 Mean Absolute Deviation

class pycast.errors.meanabsolutedeviationerror.**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

4.7.4 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

4.7.5 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 pycast.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
            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
            It represents the vlaue, after which all error values will be ignored. 90.0 for example m
            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!
```

4.8 Optimization Methods

class pycast.optimization.baseoptimizationmethod.**BaseOptimizationMethod**(*errorMeasureClass*, *precision=-1*)

Bases: object

Baseclass for all optimization methods.

__init__(*errorMeasureClass*, *precision=-1*)

Initializes the optimization method.

Parameters

- **errorMeasureClass** (*BaseErrorMeasure*) – Error measure class from pycast.errors
- **precision** (*Integer*) – Defines the accuracy for parameter tuning in 10^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 pycast.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 optimzed forecasting method with the smallest error.

Return type (BaseForecastingMethod, Dictionary)

Raise Raises a ValueError ValueError if no forecastingMethods is empty.

class pycast.optimization.gridsearch.**GridSearch**(*errorMeasureClass*, *precision=-1*)

Bases: pycast.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 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 the optimzed forecasting method, the corresponding error measure and the forecasting methods parameters.

Return type [BaseForecastingMethod, BaseErrorMeasure, Dictionary]

Raise Raises a `ValueError` 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 hte 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

CHAPTER

FIVE

USEFUL LINKS

- github Project Page
- *search*

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

E

F

G

H

I

L

M

N

O

P

S

T