tsfresh Documentation

Release 0.8.0

May 30, 2017

Contents

This is the documentation of **tsfresh**.

tsfresh is a python package. It automatically calculates a large number of time series characteristics, the so called features. Further the package contains methods to evaluate the explaining power and importance of such characteristics for regression or classification tasks.

You can jump right into the package by looking into our *Quick Start*.

CHAPTER 1

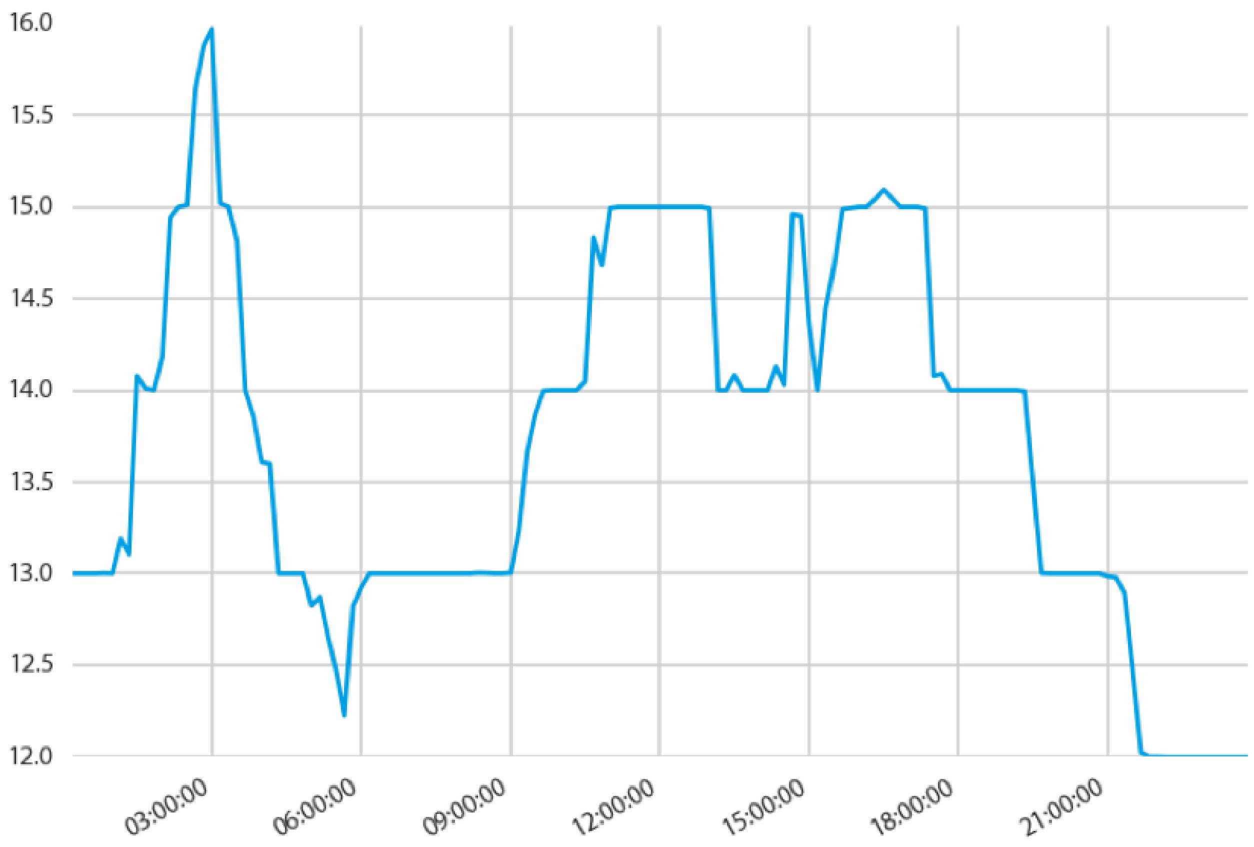
Contents

The following chapters will explain the tsfresh package in detail:

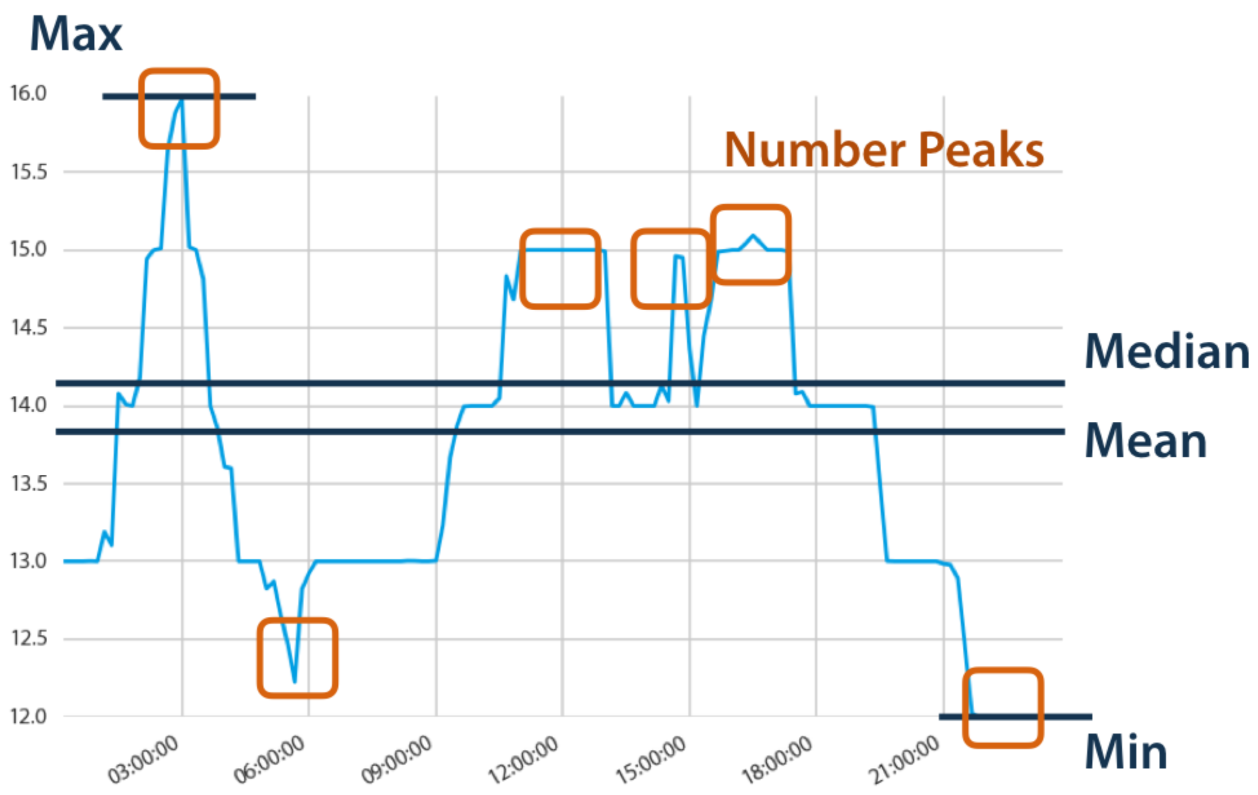
Introduction

Why do you need such a module?

tsfresh is used to to extract characteristics from time series. Let's assume you recorded the ambient temperature around your computer over one day as the following time series:



Now you want to calculate different characteristics such as the maximal or minimal temperature, the average temperature or the number of temporary temperature peaks:



Without tsfresh, you would have to calculate all those characteristics by hand. With tsfresh this process is automated and all those features can be calculated automatically.

Further tsfresh is compatible with pythons `pandas` and `scikit-learn` APIs, two important packages for Data Science endeavours in python.

What to do with these features?

The extracted features can be used to describe or cluster time series based on the extracted characteristics. Further, they can be used to build models that perform classification/regression tasks on the time series. Often the features give new insights into time series and their dynamics.

The tsfresh package has been used successfully in projects involving

- the prediction of the life span of machines
- the prediction of the quality of steel billets during a continuous casting process

What not to do with tsfresh?

Currently, tsfresh is not suitable

- for usage with streaming data
- for batch processing over a distributed architecture, where different time series are fragmented over different computational units
- to train models on the features (we do not want to reinvent the wheel, check out the python package scikit-learn for example)

However, some of these use cases could be implemented, if you have an application in mind, open an issue at https://github.com/blue-yonder/tsfresh/issues, or feel free to contact us.

What else is out there?

There is a matlab package called hctsa which can be used to automatically extract features from time series. It is also possible to use hctsa from within python by means of the pyopy package.

Quick Start

Install tsfresh

As the compiled tsfresh package is hosted on pypy you can easily install it with pip

```
pip install tsfresh
```

Dive in

Before boring yourself by reading the docs in detail, you can dive right into tsfresh with the following example:

We are given a data set containing robot failures as discussed in[1]. Each robot records time series from six different sensors. For each sample denoted by a different id we are going to classify if the robot reports a failure or not. From a machine learning point of view, our goal is to classify each group of time series.

To start, we load the data into python

```
from tsfresh.examples.robot_execution_failures import download_robot_execution_
↪failures, \
    load_robot_execution_failures
download_robot_execution_failures()
timeseries, y = load_robot_execution_failures()
```

and end up with a pandas.DataFrame *timeseries* having the following shape

	id	time	a	b	c	d	e	f
0	1	0	-1	-1	63	-3	-1	0
1	1	1	0	0	62	-3	-1	0
2	1	2	-1	-1	61	-3	0	0
3	1	3	-1	-1	63	-2	-1	0
4	1	4	-1	-1	63	-3	-1	0
...	...	...	...	...	...	...	...	...

The first column is the DataFrame index and has no meaning here. There are six different time series (a-f) for the different sensors. The different robots are denoted by the ids column.

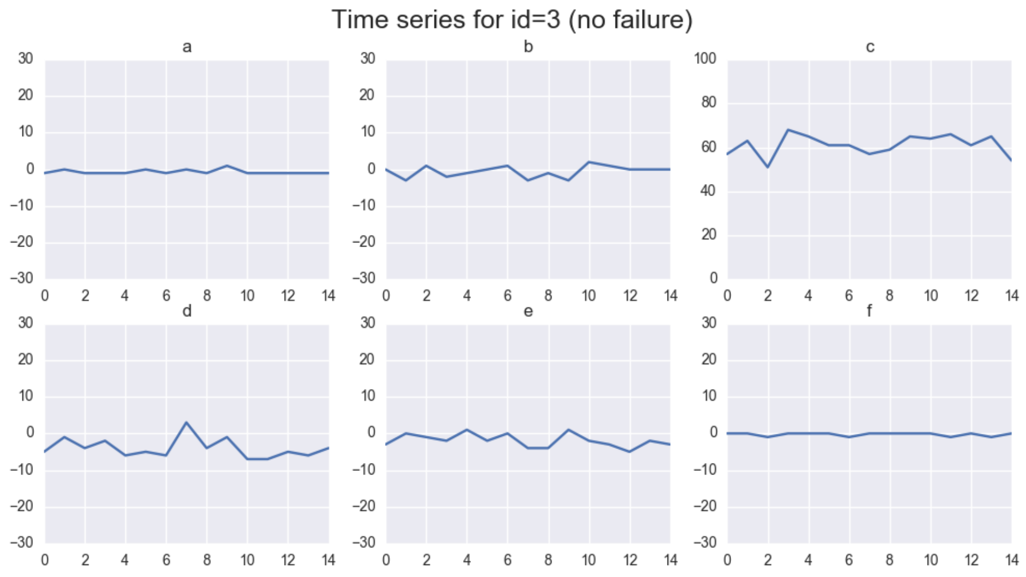
On the other hand, `y` contains the information which robot *id* reported a failure and which not:

[1] http://archive.ics.uci.edu/ml/datasets/Robot+Execution+Failures

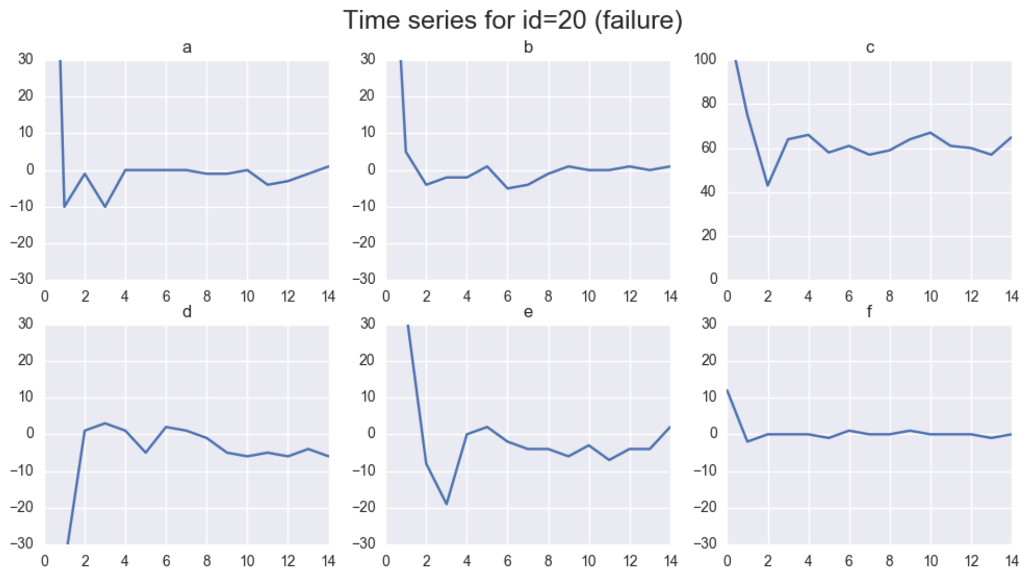
1	0
2	0
3	0
4	0
5	0
...	...

Here, for the samples with ids 1 to 5 no failure was reported.

In the following we illustrate the time series of the sample id 3 reporting no failure:



And for id 20 reporting a failure:



You can already see some differences by eye - but for successful machine learning we have to put these differences into numbers.

For this, tsfresh comes into place. It allows us to automatically extract over 1200 features from those six different time series for each robot.

For extracting all features, we do:

```
from tsfresh import extract_features
extracted_features = extract_features(timeseries, column_id="id", column_sort="time")
```

You end up with a DataFrame *extracted_features* with all more than 1200 different extracted features. We will now remove all `NaN` values (that were created by feature calculators, than can not be used on the given data, e.g. because it has too low statistics) and select only the relevant features next:

```
from tsfresh import select_features
from tsfresh.utilities.dataframe_functions import impute

impute(extracted_features)
features_filtered = select_features(extracted_features, y)
```

Only around 300 features were classified as relevant enough.

Further, you can even perform the extraction, imputing and filtering at the same time with the `tsfresh.extract_relevant_features()` function:

```
from tsfresh import extract_relevant_features

features_filtered_direct = extract_relevant_features(timeseries, y,
                                                     column_id='id', column_sort='time
↪')
```

You can now use the features contained in the DataFrame *features_filtered* (which is equal to *features_filtered_direct*) in conjunction with *y* to train your classification model. Please see the *robot_failure_example.ipynb* Jupyter Notebook in the folder named notebook for this. In this notebook a RandomForestClassifier is trained on the extracted features.

References

tsfresh

tsfresh package

Subpackages

tsfresh.convenience package

Submodules

tsfresh.convenience.relevant_extraction module

`tsfresh.convenience.relevant_extraction.`**`extract_relevant_features`**`(`*timeseries_container, y, X=None, default_fc_parameters=None, kind_to_fc_parameters=None, column_id=None, column_sort=None, column_kind=None, column_value=None, parallelization=None, show_warnings=False, disable_progressbar=False, profile=False, profiling_filename='profile.txt', profiling_sorting='cumulative', test_for_binary_target_binary_featu test_for_binary_target_real_feature= test_for_real_target_binary_feature= test_for_real_target_real_feature='k fdr_level=0.05, hypotheses_independent=False, n_processes=2, chunksize=None*`)`

High level convenience function to extract time series features from *timeseries_container*. Then return feature matrix *X* possibly augmented with relevent features with respect to target vector *y*.

For more details see the documentation of `extract_features()` and `select_features()`.

Examples

```
>>> from tsfresh.examples import load_robot_execution_failures
>>> from tsfresh import extract_relevant_features
>>> df, y = load_robot_execution_failures()
>>> X = extract_relevant_features(df, y, column_id='id', column_sort='time')
```

Parameters

- **`timeseries_container`** – The pandas.DataFrame with the time series to compute the features for, or a dictionary of pandas.DataFrames. See `extract_features()`.
- **`X`** (`pandas.DataFrame`) – A DataFrame containing additional features

- **y** (*pandas.Series*) – The target vector
- **default_fc_parameters** (*dict*) – mapping from feature calculator names to parameters. Only those names which are keys in this dict will be calculated. See the class:*ComprehensiveFCParameters* for more information.
- **kind_to_fc_parameters** (*dict*) – mapping from kind names to objects of the same type as the ones for default_fc_parameters. If you put a kind as a key here, the fc_parameters object (which is the value), will be used instead of the default_fc_parameters.
- **column_id** (*str*) – The name of the id column to group by.
- **column_sort** (*str*) – The name of the sort column.
- **column_kind** (*str*) – The name of the column keeping record on the kind of the value.
- **column_value** (*str*) – The name for the column keeping the value itself.
- **parallelization** (*str*) – Either `'per_sample'` or `'per_kind'` , see `_extract_features_parallel_per_sample()`, `_extract_features_parallel_per_kind()` and *Parallelization* for details. Choosing None makes the algorithm look for the best parallelization technique by applying some general remarks.
- **chunksize** (*None or int*) – The size of one chunk for the parallelisation
- **n_processes** (*int*) – The number of processes to use for parallelisation.
- **disable_progressbar** (*bool*) – Do not show a progressbar while doing the calculation.
- **profile** (*bool*) – Turn on profiling during feature extraction
- **profiling_sorting** (*basestring*) – How to sort the profiling results (see the documentation of the profiling package for more information)
- **profiling_filename** (*basestring*) – Where to save the profiling results.
- **test_for_binary_target_binary_feature** (*str*) – Which test to be used for binary target, binary feature (currently unused)
- **test_for_binary_target_real_feature** (*str*) – Which test to be used for binary target, real feature
- **test_for_real_target_binary_feature** (*str*) – Which test to be used for real target, binary feature (currently unused)
- **test_for_real_target_real_feature** (*str*) – Which test to be used for real target, real feature (currently unused)
- **fdr_level** (*float*) – The FDR level that should be respected, this is the theoretical expected percentage of irrelevant features among all created features.
- **hypotheses_independent** (*bool*) – Can the significance of the features be assumed to be independent? Normally, this should be set to False as the features are never independent (e.g. mean and median)
- **write_selection_report** (*bool*) – Whether to store the selection report after the Benjamini Hochberg procedure has finished.
- **result_dir** (*str*) – Where to store the selection report

Param show_warnings: Show warnings during the feature extraction (needed for debugging of calculators).

Returns Feature matrix X, possibly extended with relevant time series features.

Module contents

The `convenience` submodule contains methods that allow the user to extract and filter features conveniently.

tsfresh.examples package

Submodules

tsfresh.examples.driftbif_simulation module

`tsfresh.examples.driftbif_simulation.`**`load_driftbif`**(*n*, *l*, *m=2*, *classification=True*, *kappa_3=0.3*, *seed=False*)

Simulates n time-series with l time steps each for the m-dimensional velocity of a dissipative soliton

classification=True: target 0 means tau<=1/0.3, Dissipative Soliton with Brownian motion (purely noise driven) target 1 means tau> 1/0.3, Dissipative Soliton with Active Brownian motion (intrinsiv velocity with overlaid noise)

classification=False: target is bifurcation parameter tau

Parameters

- **n** (*int*) – number of samples
- **l** (*int*) – length of the time series
- **m** (*int*) – number of spatial dimensions (default m=2) the dissipative soliton is propagating in
- **classification** (*bool*) – distinguish between classification (default True) and regression target
- **kappa_3** (*float*) – inverse bifurcation parameter (default 0.3)
- **seed** (*float*) – random seed (default False)

Returns X, y. Time series container and target vector

Rtype X pandas.DataFrame

Rtype y pandas.DataFrame

`tsfresh.examples.driftbif_simulation.`**`sample_tau`**(*n=10*, *kappa_3=0.3*, *ratio=0.5*, *rel_increase=0.15*)

Return list of control parameters

Parameters

- **n** (*int*) – number of samples
- **kappa_3** (*float*) – inverse bifurcation point
- **ratio** (*float*) – ratio (default 0.5) of samples before and beyond drift-bifurcation
- **rel_increase** (*float*) – relative increase from bifurcation point

Returns tau. List of sampled bifurcation parameter

Rtype tau list

class `tsfresh.examples.driftbif_simulation.`**`velocity`**(*tau=3.8*, *kappa_3=0.3*, *Q=1950.0*, *R=0.0003*, *delta_t=0.05*, *seed=None*)

Bases: `object`

Simulates the velocity of a dissipative soliton (kind of self organized particle). The equilibrium velocity without noise R=0 for $ au>1.0/kappa_3$ is $kappa_3 sqrt{(tau - 1.0/kappa_3)/Q}. Before the drift-bifurcation $ au le 1.0/kappa_3$ the velocity is zero.

References

```
>>> ds = velocity(tau=3.5) # Dissipative soliton with equilibrium velocity 1.5e-3
>>> print(ds.label) # Discriminating before or beyond Drift-Bifurcation
1
>>> print(ds.deterministic) # Equilibrium velocity
0.0015191090506254991
>>> v = ds.simulate(20000) #Simulated velocity as a time series with 20000 time␣
↪steps being disturbed by Gaussian white noise
```

`simulate`(*N*, *v0=array([0., 0.])*)

Parameters

- **`N`** (*`int`*) – number of time steps
- **`v0`** (*`ndarray`*) – initial velocity vector

Returns time series of velocity vectors with shape (N, v0.shape[0])

Return type ndarray

tsfresh.examples.har_dataset module

This module implements functions to download and load the Human Activity Recognition dataset [4]. A description of the data set can be found in [5].

References

`tsfresh.examples.har_dataset.`**`download_har_dataset`**()

Download human activity recognition dataset from UCI ML Repository and store it at /tsfresh/notebooks/data.

Examples

```
>>> from tsfresh.examples import har_dataset
>>> download_har_dataset()
```

`tsfresh.examples.har_dataset.`**`load_har_classes`**()

`tsfresh.examples.har_dataset.`**`load_har_dataset`**()

tsfresh.examples.robot_execution_failures module

This module implements functions to download the Robot Execution Failures LP1 Data Set[1] and load it as as DataFrame.

Important: You need to download the data set yourself, either manually or via the function `download_robot_execution_failures()`

References

`tsfresh.examples.robot_execution_failures.`**`download_robot_execution_failures`**`()`

Download the Robot Execution Failures LP1 Data Set[1] from the UCI Machine Learning Repository[2] and store it locally. :return:

Examples

```
>>> from tsfresh.examples import download_robot_execution_failures
>>> download_robot_execution_failures()
```

`tsfresh.examples.robot_execution_failures.`**`load_robot_execution_failures`**`()`

Load the Robot Execution Failures LP1 Data Set[1]. The Time series are passed as a flat DataFrame.

Examples

```
>>> from tsfresh.examples import load_robot_execution_failures
>>> df, y = load_robot_execution_failures()
>>> print(df.shape)
(1320, 8)
```

Returns time series data as `pandas.DataFrame` and target vector as `pandas.Series`

Return type tuple

tsfresh.examples.test_tsfresh_baseline_dataset module

This module implements a function to download a json timeseries data set that is utilised by tests/baseline/tsfresh_features_test.py to test calculated feature names and their calculated values are consistent with the known baseline.

`tsfresh.examples.test_tsfresh_baseline_dataset.`**`download_json_dataset`**`()`

Download the tests baseline timeseries json data set and store it at tsfresh/examples/data/test_tsfresh_baseline_dataset/data.json.

Examples

```
>>> from tsfresh.examples import test_tsfresh_baseline_dataset
>>> download_json_dataset()
```

Module contents

Module with exemplary data sets to play around with.

See for eample the *Quick Start* section on how to use them.

tsfresh.feature_extraction package

Submodules

tsfresh.feature_extraction.extraction module

This module contains the main function to interact with tsfresh: extract features

`tsfresh.feature_extraction.extraction.`**`extract_features`**(*timeseries_container, default_fc_parameters=None, kind_to_fc_parameters=None, column_id=None, column_sort=None, column_kind=None, column_value=None, parallelization=None, chunksize=None, n_processes=2, show_warnings=False, disable_progressbar=False, impute_function=None, profile=False, profiling_filename='profile.txt', profiling_sorting='cumulative'*)

Extract features from

- a `pandas.DataFrame` containing the different time series

or

- a dictionary of `pandas.DataFrame` each containing one type of time series

In both cases a `pandas.DataFrame` with the calculated features will be returned.

For a list of all the calculated time series features, please see the *`ComprehensiveFCParameters`* class, which is used to control which features with which parameters are calculated.

For a detailed explanation of the different parameters and data formats please see *Data Formats*.

Examples

```
>>> from tsfresh.examples import load_robot_execution_failures
>>> from tsfresh import extract_features
>>> df, _ = load_robot_execution_failures()
>>> X = extract_features(df, column_id='id', column_sort='time')
```

Parameters

- **timeseries_container** (*pandas.DataFrame or dict*) – The pandas.DataFrame with the time series to compute the features for, or a dictionary of pandas.DataFrames.
- **default_fc_parameters** (*dict*) – mapping from feature calculator names to parameters. Only those names which are keys in this dict will be calculated. See the class:*ComprehensiveFCParameters* for more information.
- **kind_to_fc_parameters** (*dict*) – mapping from kind names to objects of the same type as the ones for default_fc_parameters. If you put a kind as a key here, the fc_parameters object (which is the value), will be used instead of the default_fc_parameters.
- **column_id** (*str*) – The name of the id column to group by.
- **column_sort** (*str*) – The name of the sort column.
- **column_kind** (*str*) – The name of the column keeping record on the kind of the value.
- **column_value** (*str*) – The name for the column keeping the value itself.
- **parallelization** (*str*) – Either `'per_sample'` or `'per_kind'` , see `_extract_features_parallel_per_sample()`, `_extract_features_parallel_per_kind()` and *Parallelization* for details. Choosing None makes the algorithm look for the best parallelization technique by applying some general assumptions.
- **chunksize** (*None or int*) – The size of one chunk for the parallelisation
- **n_processes** (*int*) – The number of processes to use for parallelisation.
- **disable_progressbar** (*bool*) – Do not show a progressbar while doing the calculation.
- **impute_function** (*None or function*) – None, if no imputing should happen or the function to call for imputing.
- **profile** (*bool*) – Turn on profiling during feature extraction
- **profiling_sorting** (*basestring*) – How to sort the profiling results (see the documentation of the profiling package for more information)
- **profiling_filename** (*basestring*) – Where to save the profiling results.

Param show_warnings: Show warnings during the feature extraction (needed for debugging of calculators).

Returns The (maybe imputed) DataFrame containing extracted features.

Return type pandas.DataFrame

tsfresh.feature_extraction.feature_calculators module

This module contains the feature calculators that take time series as input and calculate the values of the feature. There are three types of features:

1. aggregate features without parameter
2. aggregate features with parameter
3. apply features with parameters

While type 1 and 2 are designed to be used with pandas aggregate, they will only return one singular feature. To not unnecessarily redo auxiliary calculations, in type 3 a group of features is calculated at the same time. They can be used with pandas apply.

`tsfresh.feature_extraction.feature_calculators.`**`abs_energy`**(*x*, **arg*, ***args*)

Returns the absolute energy of the time series which is the sum over the squared values

$$E = \sum_{i=1,\ldots,n} x_i^2$$

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`absolute_sum_of_changes`**(*x*, **arg*, ***args*)

Returns the sum over the absolute value of consecutive changes in the series x

$$\sum_{i=1,\ldots,n-1} \mid x_{i+1} - x_i \mid$$

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`approximate_entropy`**(*x*, *m*, *r*)

Implements a vectorized Approximate entropy algorithm.

https://en.wikipedia.org/wiki/Approximate_entropy

For short time-series this method is highly dependent on the parameters, but should be stable for N > 2000, see:

Yentes et al. (2012) - *The Appropriate Use of Approximate Entropy and Sample Entropy with Short Data Sets*

Other shortcomings and alternatives discussed in:

Richman & Moorman (2000) - *Physiological time-series analysis using approximate entropy and sample entropy*

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **m** (*int*) – Length of compared run of data
- **r** (*float*) – Filtering level, must be positive

Returns Approximate entropy

Return type float

This function is of type: aggregate_with_parameters `tsfresh.feature_extraction.feature_calculators.`**`ar_coefficient`**(*x*, **arg*, ***args*)

This feature calculator fits the unconditional maximum likelihood of an autoregressive AR(k) process. The k parameter is the maximum lag of the process

$$X_t = \varphi_0 + \sum_{i=1}^{k} \varphi_i X_{t-i} + \varepsilon_t$$

For the configurations from param which should contain the maxlag "k" and such an AR process is calculated. Then the coefficients φ_i whose index i contained from "coeff" are returned.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **c** (*str*) – the time series name
- **param** (*list*) – contains dictionaries {"coeff": x, "k": y} with x,y int

Return x the different feature values

Return type pandas.Series

This function is of type: apply

`tsfresh.feature_extraction.feature_calculators.`**`augmented_dickey_fuller`**(*x*, **arg*, ***args*)

The Augmented Dickey-Fuller test is a hypothesis test which checks whether a unit root is present in a time series sample. This feature calculator returns the value of the respective test statistic.

See the statsmodels implementation for references and more details.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`autocorrelation`**(*x*, **arg*, ***args*)

Calculates the lag autocorrelation of a lag value of lag.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **lag** (*int*) – the lag

Returns the value of this feature

Return type float

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`binned_entropy`**(*x*, **arg*, ***args*)

First bins the values of x into max_bins equidistant bins. Then calculates the value of

$$- \sum_{k=0}^{min(max\_bins, len(x))} p_k log(p_k) \cdot \mathbf{1}_{(p_k > 0)}$$

where p_k is the percentage of samples in bin k.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **max_bins** (*int*) – the maximal number of bins

Returns the value of this feature

Return type float

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`count_above_mean`**(*x*, **arg*, ***args*)

Returns the number of values in x that are higher than the mean of x

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`count_below_mean`**(*x*, **arg*, ***args*)

Returns the number of values in x that are lower than the mean of x

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`cwt_coefficients`**(*x*, **arg*, ***args*)

Calculates a Continuous wavelet transform for the Ricker wavelet, also known as the “Mexican hat wavelet” which is defined by

$$\frac{2}{\sqrt{3a}\pi^{\frac{1}{4}}}(1 - \frac{x^2}{a^2})exp(-\frac{x^2}{2a^2})$$

where a is the width parameter of the wavelet function.

This feature calculator takes three different parameter: widths, coeff and w. The feature calculater takes all the different widths arrays and then calculates the cwt one time for each different width array. Then the values for the different coefficient for coeff and width w are returned. (For each dic in param one feature is returned)

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **c** (*str*) – the time series name
- **param** (*list*) – contains dictionaries {“widths”:x, “coeff”: y, “w”: z} with x array of int and y,z int

Returns the different feature values

Return type pandas.Series

This function is of type: apply

`tsfresh.feature_extraction.feature_calculators.`**`fft_coefficient`**(*x*, **arg*, ***args*)

Calculates the fourier coefficients of the one-dimensional discrete Fourier Transform for real input by fast fourier transformation algorithm

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **c** (*str*) – the time series name
- **param** (*list*) – contains dictionaries {"coeff": x} with x int and x >= 0

Returns the different feature values

Return type pandas.Series

This function is of type: apply

`tsfresh.feature_extraction.feature_calculators.`**`first_location_of_maximum`**(*x*, **arg*, ***args*)

Returns the first location of the maximum value of x. The position is calculated relatively to the length of x.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`first_location_of_minimum`**(*x*, **arg*, ***args*)

Returns the first location of the minimal value of x. The position is calculated relatively to the length of x.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`friedrich_coefficients`**(*x*, *c*, *param*)

Coefficients of polynomial $h(x)$, which has been fitted to the deterministic dynamics of Langevin model .. math:

Unexpected indentation.

```
\dot{x}(t) = h(x(t)) + \mathcal{N}(0,R)
```

as described by

Friedrich et al. (2000): Physics Letters A 271, p. 217-222 *Extracting model equations from experimental data*

For short time-series this method is highly dependent on the parameters.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **c** (*str*) – the time series name
- **param** (*list*) – contains dictionaries {"coeff": x} with x int and x >= 0

Returns the different feature values

Return type pandas.Series

This function is of type: apply

`tsfresh.feature_extraction.feature_calculators.`**`has_duplicate`**(*x*, **arg*, ***args*)

Checks if any value in x occurs more than once

Parameters **`x`** (*`pandas.Series`*) – the time series to calculate the feature of

Returns the value of this feature

Return type bool

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`has_duplicate_max`**(*x*, **arg*, ***args*)

Checks if the maximum value of x is observed more than once

Parameters **`x`** (*`pandas.Series`*) – the time series to calculate the feature of

Returns the value of this feature

Return type bool

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`has_duplicate_min`**(*x*, **arg*, ***args*)

Checks if the minimal value of x is observed more than once

Parameters **`x`** (*`pandas.Series`*) – the time series to calculate the feature of

Returns the value of this feature

Return type bool

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`index_mass_quantile`**(*x*, **arg*, ***args*)

Those apply features calculate the relative index i where q% of the mass of the time series x lie left of i. For example for q = 50% this feature calculator will return the mass center of the time series

Parameters

- **`x`** (*`pandas.Series`*) – the time series to calculate the feature of
- **`c`** (*`str`*) – the time series name
- **`param`** (*`list`*) – contains dictionaries {"q": x} with x float

Returns the different feature values

Return type pandas.Series

This function is of type: apply

`tsfresh.feature_extraction.feature_calculators.`**`kurtosis`**(*x*, **arg*, ***args*)

Returns the kurtosis of x (calculated with the adjusted Fisher-Pearson standardized moment coefficient G2).

Parameters **`x`** (*`pandas.Series`*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`large_number_of_peaks`**(*x*, **arg*, ***args*)

Checks if the number of peaks is higher than n.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **n** (*int*) – the number of peaks to compare

Returns the value of this feature

Return type bool

This function is of type: aggregate_with_parameters

tsfresh.feature_extraction.feature_calculators.**large_standard_deviation**(*x*, **arg*, ***args*)

Boolean variable denoting if the standard dev of x is higher than 'r' times the range = difference between max and min of x. Hence it checks if

$$std(x) > r * (max(X) - min(X))$$

According to a rule of the thumb, the standard deviation should be a forth of the range of the values.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **r** (*float*) – the percentage of the range to compare with

Returns the value of this feature

Return type bool

This function is of type: aggregate_with_parameters

tsfresh.feature_extraction.feature_calculators.**last_location_of_maximum**(*x*, **arg*, ***args*)

Returns the relative last location of the maximum value of x. The position is calculated relatively to the length of x.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

tsfresh.feature_extraction.feature_calculators.**last_location_of_minimum**(*x*, **arg*, ***args*)

Returns the last location of the minimal value of x. The position is calculated relatively to the length of x.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

tsfresh.feature_extraction.feature_calculators.**length**(*x*, **arg*, ***args*)

Returns the length of x

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type int

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`longest_strike_above_mean`**(*x*, **arg*, ***args*)

Returns the length of the longest consecutive subsequence in x that is bigger than the mean of x

Parameters **x** (`pandas.Series`) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`longest_strike_below_mean`**(*x*, **arg*, ***args*)

Returns the length of the longest consecutive subsequence in x that is smaller than the mean of x

Parameters **x** (`pandas.Series`) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`max_langevin_fixed_point`**(*x*, *r*, *m*)

Largest fixed point of dynamics :math:argmax_x {h(x)=0}‘ estimated from polynomial $h(x)$, which has been fitted to the deterministic dynamics of Langevin model .. math:

Unexpected indentation.

```
\dot(x)(t) = h(x(t)) + R \mathcal(N)(0,1)
```

as described by

Friedrich et al. (2000): Physics Letters A 271, p. 217-222 *Extracting model equations from experimental data*

For short time-series this method is highly dependent on the parameters.

Parameters

- **x** (`pandas.Series`) – the time series to calculate the feature of
- **m** (`int`) – order of polynom to fit for estimating fixed points of dynamics
- **r** (`float`) – number of quantils to use for averaging

Returns Largest fixed point of deterministic dynamics

Return type float

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`maximum`**(*x*)

Calculates the highest value of the time series x.

Parameters **x** (`pandas.Series`) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`mean`**(*x*)

Returns the mean of x

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`mean_abs_change`**(*x*, **arg*, ***args*)

Returns the mean over the absolute differences between subsequent time series values which is

$$\frac{1}{n}\sum_{i=1,\ldots,n-1}|x_{i+1}-x_i|$$

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`mean_abs_change_quantiles`**(*x*, **arg*, ***args*)

First fixes a corridor given by the quantiles ql and qh of the distribution of x. Then calculates the average absolute value of consecutive changes of the series x inside this corridor. Think about selecting a corridor on the y-Axis and only calculating the mean of the absolute change of the time series inside this corridor.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **ql** (*float*) – the lower quantile of the corridor
- **qh** (*float*) – the higher quantile of the corridor

Returns the value of this feature

Return type float

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`mean_autocorrelation`**(*x*, **arg*, ***args*)

Calculates the average autocorrelation (Compare to http://en.wikipedia.org/wiki/Autocorrelation#Estimation), taken over different all possible lags (1 to length of x)

$$\frac{1}{n}\sum_{l=1,\ldots,n}\frac{1}{(n-l)\sigma^2}\sum_{t=1}^{n-l}(X_t-\mu)(X_{t+l}-\mu)$$

where n is the length of the time series X_i, σ^2 its variance and μ its mean.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`mean_change`**(*x*, **arg*, ***args*)

Returns the mean over the absolute differences between subsequent time series values which is

$$\frac{1}{n} \sum_{i=1,\ldots,n-1} x_{i+1} - x_i$$

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`mean_second_derivate_central`**(*x*, **arg*, ***args*)

Returns the mean value of a central approximation of the second derivative

$$\frac{1}{n} \sum_{i=1,\ldots,n-1} \frac{1}{2}(x_{i+2} - 2 \cdot x_{i+1} + x_i)$$

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`median`**(*x*)

Returns the median of x

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`minimum`**(*x*)

Calculates the lowest value of the time series x.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`not_apply_to_raw_numbers`**(*func*)

This decorator makes sure that the function func is only called on objects that are not numbers.Number

Parameters **func** – the method that should only be executed on objects which are not a numbers.Number

Returns the decorated version of func which returns 0 if the first argument x is a numbers.Number. For every other x the output of func is returned `tsfresh.feature_extraction.feature_calculators.`**`number_cwt_peaks`**(*x*, **arg*, ***args*)

This feature calculator searches for different peaks in x. To do so, x is smoothed by a ricker wavelet and for widths ranging from 1 to n. This feature calculator returns the number of peaks that occur at enough width scales and with sufficiently high Signal-to-Noise-Ratio (SNR)

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **n** (*int*) – maximum width to consider

Returns the value of this feature

Return type int

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`number_peaks`**(*x*, **arg*, ***args*)

Calculates the number of peaks of at least support n in the time series x. A peak of support n is defined as a subsequence of x where a value occurs, which is bigger than its n neighbours to the left and to the right.

Hence in the sequence

```
>>> x = [3, 0, 0, 4, 0, 0, 13]
```

4 is a peak of support 1 and 2 because in the subsequences

```
>>> [0, 4, 0]
>>> [0, 0, 4, 0, 0]
```

4 is still the highest value. Here, 4 is not a peak of support 3 because 13 is the 3th neighbour to the right of 4 and its bigger than 4.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **n** (*int*) – the support of the peak

Returns the value of this feature

Return type float

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`percentage_of_reoccurring_datapoints_to_all_`**

Returns the percentage of unique values, that are present in the time series more than once.

len(different values occurring more than once) / len(different values)

This means the percentage is normalized to the number of unique values, in contrast to the percentage_of_reoccurring_values_to_all_values.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`percentage_of_reoccurring_values_to_all_valu`**

Returns the ratio of unique values, that are present in the time series more than once.

# of data points occurring more than once / # of all data points

This means the ratio is normalized to the number of data points in the time series, in contrast to the percentage_of_reoccurring_datapoints_to_all_datapoints.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`quantile`**(*x*, **arg*, ***args*)

Calculates the q quantile of x. This is the value of x greater than q% of the ordered values from x.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **q** (*float*) – the quantile to calculate

Returns the value of this feature

Return type float

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`range_count`**(*x*, *min*, *max*)

Count observed values within the interval [min, max).

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **min** (*int or float*) – the inclusive lower bound of the range
- **max** (*int or float*) – the exclusive upper bound of the range

Returns the count of values within the range

Return type int

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`ratio_value_number_to_time_series_length`**(*x*, **ar* ***a*

Returns a factor which is 1 if all values in the time series occur only once, and below one if this is not the case. In principle, it just returns

# unique values / # values

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`sample_entropy`**(*x*)

Calculate and return sample entropy of x. References: ——— [1] http://en.wikipedia.org/wiki/Sample_Entropy [2] https://www.ncbi.nlm.nih.gov/pubmed/10843903?dopt=Abstract

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **tolerance** (*float*) – normalization factor; equivalent to the common practice of expressing the tolerance as r times the standard deviation

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`set_property`**(*key*, *value*)

This method returns a decorator that sets the property key of the function to value

`tsfresh.feature_extraction.feature_calculators.`**`skewness`**(*x*, **arg*, ***args*)

Returns the sample skewness of x (calculated with the adjusted Fisher-Pearson standardized moment coefficient G1).

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`spkt_welch_density`**(*x*, **arg*, ***args*)

This feature calculator estimates the cross power spectral density of the time series x at different frequencies. To do so, the time series is first shifted from the time domain to the frequency domain.

The feature calculators returns the power spectrum of the different frequencies.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **c** (*str*) – the time series name
- **param** (*list*) – contains dictionaries {“coeff”: x} with x int

Returns the different feature values

Return type pandas.Series

This function is of type: apply

`tsfresh.feature_extraction.feature_calculators.`**`standard_deviation`**(*x*, **arg*, ***args*)

Returns the standard deviation of x

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`sum_of_reoccurring_data_points`**(*x*, **arg*, ***args*)

Returns the sum of all data points, that are present in the time series more than once.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`sum_of_reoccurring_values`**(*x*, **arg*, ***args*)

Returns the sum of all values, that are present in the time series more than once.

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`sum_values`**(*x*)

Calculates the sum over the time series values

Parameters **x** (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type bool

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`symmetry_looking`**(*x*, **arg*, ***args*)

Boolean variable denoting if the distribution of x *looks symmetric*. This is the case if

$$|mean(X) - median(X)| < r * (max(X) - min(X))$$

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **r** (*float*) – the percentage of the range to compare with

Returns the value of this feature

Return type bool

This function is of type: apply

`tsfresh.feature_extraction.feature_calculators.`**`time_reversal_asymmetry_statistic`**(*x*, **arg*, ***args*)

This function calculates the value of

$$\frac{1}{n - 2lag} \sum_{i=0}^{n-2lag} x_{i+2 \cdot lag}^2 \cdot x_{i+lag} - x_{i+lag} \cdot x_i^2$$

which is

$$\mathbb{E}[L^2(X)^2 \cdot L(X) - L(X) \cdot X^2]$$

where $\mathbb{E}$ is the mean and L is the lag operator. It was proposed in [1] as a promising feature to extract from time series.

References

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **lag** (*int*) – the lag that should be used in the calculation of the feature

Returns the value of this feature

Return type float

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`value_count`**(*x*, *value*)

Count occurrences of *value* in time series x.

Parameters

- **x** (*pandas.Series*) – the time series to calculate the feature of
- **value** (*int or float*) – the value to be counted

Returns the count

Return type int

This function is of type: aggregate_with_parameters

`tsfresh.feature_extraction.feature_calculators.`**`variance`**(*x*, **arg*, ***args*)

Returns the variance of x

Parameters x (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type float

This function is of type: aggregate

`tsfresh.feature_extraction.feature_calculators.`**`variance_larger_than_standard_deviation`**(*x*, **arg*, ***ar*

Boolean variable denoting if the variance of x is greater than its standard deviation. Is equal to variance of x being larger than 1

Parameters x (*pandas.Series*) – the time series to calculate the feature of

Returns the value of this feature

Return type bool

This function is of type: aggregate

tsfresh.feature_extraction.settings module

This file contains methods/objects for controlling which features will be extracted when calling extract_features. For the naming of the features, see *Feature Calculation*.

class tsfresh.feature_extraction.settings.**ComprehensiveFCParameters**

Bases: dict

Create a new ComprehensiveFCParameters instance. You have to pass this instance to the extract_feature instance.

It is basically a dictionary (and also based on one), which is a mapping from string (the same names that are in the feature_calculators.py file) to a list of dictionary of parameters, which should be used when the function with this name is called.

Only those strings (function names), that are keys in this dictionary, will be later used to extract features - so whenever you delete a key from this dict, you disable the calculation of this feature.

You can use the settings object with

```
>>> from tsfresh.feature_extraction import extract_features,␣
↪ComprehensiveFCParameters
>>> extract_features(df, default_fc_parameters=ComprehensiveFCParameters())
```

to extract all features (which is the default nevertheless) or you change the ComprehensiveFCParameters object to other types (see below).

class tsfresh.feature_extraction.settings.**EfficientFCParameters**

Bases: *tsfresh.feature_extraction.settings.ComprehensiveFCParameters*

This class is a child class of the ComprehensiveFCParameters class and has the same functionality as its base class.

The only difference is, that the features with high computational costs are not calculated. Those are denoted by the attribute "high_comp_cost".

You should use this object when calling the extract function, like so:

```
>>> from tsfresh.feature_extraction import extract_features, EfficientFCParameters
>>> extract_features(df, default_fc_parameters=EfficientFCParameters())
```

class tsfresh.feature_extraction.settings.**MinimalFCParameters**

Bases: *tsfresh.feature_extraction.settings.ComprehensiveFCParameters*

This class is a child class of the ComprehensiveFCParameters class and has the same functionality as its base class. The only difference is, that most of the feature calculators are disabled and only a small subset of calculators will be calculated at all. Those are donated by an attribute called "minimal".

Use this class for quick tests of your setup before calculating all features which could take some time depending of your data set size.

You should use this object when calling the extract function, like so:

```
>>> from tsfresh.feature_extraction import extract_features, MinimalFCParameters
>>> extract_features(df, default_fc_parameters=MinimalFCParameters())
```

tsfresh.feature_extraction.settings.**from_columns**(*columns*)

Creates a mapping from kind names to fc_parameters objects (which are itself mappings from feature calculators to settings) to extract only the features contained in the columns. To do so, for every feature name in columns this method

1.split the column name into col, feature, params part

2.decide which feature we are dealing with (aggregate with/without params or apply)

3.add it to the new name_to_function dict

4.set up the params

Parameters **`columns`** (*`list of str`*) – containing the feature names

Returns The kind_to_fc_parameters object ready to be used in the extract_features function.

Return type dict

`tsfresh.feature_extraction.settings.`**`get_aggregate_functions`**(*fc_parameters*, *column_prefix*)

Returns a dictionary with some of the column name mapped to the feature calculators that are specified in the fc_parameters. This dictionary includes those calculators, that can be used in a pandas group by command to extract all aggregate features at the same time.

Parameters

- **`fc_parameters`** (`ComprehensiveFCParameters` *`or child class`*) – mapping from feature calculator names to settings.
- **`column_prefix`** (*`basestring`*) – the prefix for all column names.

Returns mapping of column name to function calculator

Return type dict

`tsfresh.feature_extraction.settings.`**`get_apply_functions`**(*fc_parameters*, *column_prefix*)

Returns a dictionary with some of the column name mapped to the feature calculators that are specified in the fc_parameters. This dictionary includes those calculators, that can *not* be used in a pandas group by command to extract all aggregate features at the same time.

Parameters

- **`fc_parameters`** (`ComprehensiveFCParameters` *`or child class`*) – mapping from feature calculator names to settings.
- **`column_prefix`** (*`basestring`*) – the prefix for all column names.

Returns all functions to use for feature extraction

Return type list

Module contents

The *`tsfresh.feature_extraction`* module contains methods to extract the features from the time series

tsfresh.feature_selection package

Submodules

tsfresh.feature_selection.feature_selector module

Contains a feature selection method that evaluates the importance of the different extracted features. To do so, for every feature the influence on the target is evaluated by an univariate tests and the p-Value is calculated. The methods that calculate the p-values are called feature selectors.

Afterwards the Benjamini Hochberg procedure which is a multiple testing procedure decides which features to keep and which to cut off (solely based on the p-values).

`tsfresh.feature_selection.feature_selector.`**`benjamini_hochberg_test`**(*df_pvalues*, *hypotheses_independent*, *fdr_level*)

This is an implementation of the benjamini hochberg procedure that calculates which of the hypotheses belonging to the different p-Values from df_p to reject. While doing so, this test controls the false discovery rate, which is the ratio of false rejections by all rejections:

$$FDR = \mathbb{E}\left[\frac{|\text{false rejections}|}{|\text{all rejections}|}\right]$$

References

Parameters

- **`df_pvalues`** (*`pandas.DataFrame`*) – This DataFrame should contain the p_values of the different hypotheses in a column named "p_values".
- **`hypotheses_independent`** (*`bool`*) – Can the significance of the features be assumed to be independent? Normally, this should be set to False as the features are never independent (e.g. mean and median)
- **`fdr_level`** (*`float`*) – The FDR level that should be respected, this is the theoretical expected percentage of irrelevant features among all created features.

Returns The same DataFrame as the input, but with an added boolean column "rejected".

Return type pandas.DataFrame

`tsfresh.feature_selection.feature_selector.`**`check_fs_sig_bh`**(*X*, *y*, *n_processes=2*, *chunksize=None*, *fdr_level=0.05*, *hypotheses_independent=False*, *test_for_binary_target_real_feature='mann'*)

The wrapper function that calls the significance test functions in this package. In total, for each feature from the input pandas.DataFrame an univariate feature significance test is conducted. Those tests generate p values that are then evaluated by the Benjamini Hochberg procedure to decide which features to keep and which to delete.

We are testing

H_0 = the Feature is not relevant and can not be added

against

H_1 = the Feature is relevant and should be kept

or in other words

H_0 = Target and Feature are independent / the Feature has no influence on the target

H_1 = Target and Feature are associated / dependent

When the target is binary this becomes

$H_0 = (F_{\text{target}=1} = F_{\text{target}=0})$

$H_1 = (F_{\text{target}=1} \neq F_{\text{target}=0})$

Where F is the distribution of the target.

In the same way we can state the hypothesis when the feature is binary

$$H_0 = (T_{\text{feature}=1} = T_{\text{feature}=0})$$

$$H_1 = (T_{\text{feature}=1} \neq T_{\text{feature}=0})$$

Here T is the distribution of the target.

TODO: And for real valued?

Parameters

- **X** (*pandas.DataFrame*) – The DataFrame containing all the features and the target
- **y** (*pandas.Series*) – The target vector
- **test_for_binary_target_real_feature** (*str*) – Which test to be used for binary target, real feature
- **fdr_level** (*float*) – The FDR level that should be respected, this is the theoretical expected percentage of irrelevant features among all created features.
- **hypotheses_independent** (*bool*) – Can the significance of the features be assumed to be independent? Normally, this should be set to False as the features are never independent (e.g. mean and median)
- **n_processes** (*int*) – Number of processes to use during the p-value calculation
- **chunksize** (*int*) – Size of the chunks submitted to the worker processes

Returns A pandas.DataFrame with each column of the input DataFrame X as index with information on the significance of this particular feature. The DataFrame has the columns "Feature", "type" (binary, real or const), "p_value" (the significance of this feature as a p-value, lower means more significant) "rejected" (if the Benjamini Hochberg procedure rejected this feature)

Return type pandas.DataFrame

tsfresh.feature_selection.selection module

This module contains the filtering process for the extracted features. The filtering procedure can also be used on other features that are not based on time series.

`tsfresh.feature_selection.selection.`**`select_features`**(*X, y, test_for_binary_target_binary_feature='fisher', test_for_binary_target_real_feature='mann', test_for_real_target_binary_feature='mann', test_for_real_target_real_feature='kendall', fdr_level=0.05, hypotheses_independent=False, n_processes=2, chunksize=None*)

Check the significance of all features (columns) of feature matrix X and return a possibly reduced feature matrix only containing relevant features.

The feature matrix must be a pandas.DataFrame in the format:

index	feature_1	feature_2	...	feature_N
A	...	...	...	...
B	...	...	...	...
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...

Each column will be handled as a feature and tested for its significance to the target.

The target vector must be a pandas.Series or numpy.array in the form

index	target
A	...
B	...
.	...
.	...

and must contain all id's that are in the feature matrix. If y is a numpy.array without index, it is assumed that y has the same order and length than X and the rows correspond to each other.

Examples

```
>>> from tsfresh.examples import load_robot_execution_failures
>>> from tsfresh import extract_features, select_features
>>> df, y = load_robot_execution_failures()
>>> X_extracted = extract_features(df, column_id='id', column_sort='time')
>>> X_selected = select_features(X_extracted, y)
```

Parameters

- **X** (*pandas.DataFrame*) – Feature matrix in the format mentioned before which will be reduced to only the relevant features. It can contain both binary or real-valued features at the same time.
- **y** (*pandas.Series or numpy.ndarray*) – Target vector which is needed to test which features are relevant. Can be binary or real-valued.
- **test_for_binary_target_binary_feature** (*str*) – Which test to be used for binary target, binary feature (currently unused)
- **test_for_binary_target_real_feature** (*str*) – Which test to be used for binary target, real feature
- **test_for_real_target_binary_feature** (*str*) – Which test to be used for real target, binary feature (currently unused)
- **test_for_real_target_real_feature** (*str*) – Which test to be used for real target, real feature (currently unused)
- **fdr_level** (*float*) – The FDR level that should be respected, this is the theoretical expected percentage of irrelevant features among all created features.
- **hypotheses_independent** (*bool*) – Can the significance of the features be assumed to be independent? Normally, this should be set to False as the features are never independent (e.g. mean and median)
- **n_processes** (*int*) – Number of processes to use during the p-value calculation
- **chunksize** (*int*) – Size of the chunks submitted to the worker processes

Returns The same DataFrame as X, but possibly with reduced number of columns (= features).

Return type pandas.DataFrame

Raises `ValueError` when the target vector does not fit to the feature matrix.

tsfresh.feature_selection.significance_tests module

Contains the methods from the following paper about FRESH [2]

Fresh is based on hypothesis tests that individually check the significance of every generated feature on the target. It makes sure that only features are kept, that are relevant for the regression or classification task at hand. FRESH decide between four settings depending if the features and target are binary or not.

The four functions are named

1. `target_binary_feature_binary_test()`: Target and feature are both binary
2. `target_binary_feature_real_test()`: Target is binary and feature real
3. `target_real_feature_binary_test()`: Target is real and the feature is binary
4. `target_real_feature_real_test()`: Target and feature are both real

References

`tsfresh.feature_selection.significance_tests.`**`target_binary_feature_binary_test`**(*x*, *y*)

Calculate the feature significance of a binary feature to a binary target as a p-value. Use the two-sided univariate fisher test from `fisher_exact()` for this.

Parameters

- **x** (*pandas.Series*) – the binary feature vector
- **y** (*pandas.Series*) – the binary target vector

Returns the p-value of the feature significance test. Lower p-values indicate a higher feature significance

Return type float

Raise `ValueError` if the target or the feature is not binary.

`tsfresh.feature_selection.significance_tests.`**`target_binary_feature_real_test`**(*x*, *y*, *test*)

Calculate the feature significance of a real-valued feature to a binary target as a p-value. Use either the *Mann-Whitney U* or *Kolmogorov Smirnov* from `mannwhitneyu()` or `ks_2samp()` for this.

Parameters

- **x** (*pandas.Series*) – the real-valued feature vector
- **y** (*pandas.Series*) – the binary target vector
- **test** (*str*) – The significance test to be used. Either `'mann'` for the Mann-Whitney-U test or `'smir'` for the Kolmogorov-Smirnov test

Returns the p-value of the feature significance test. Lower p-values indicate a higher feature significance

Return type float

Raise `ValueError` if the target is not binary.

`tsfresh.feature_selection.significance_tests.`**`target_real_feature_binary_test`**(*x*, *y*)

Calculate the feature significance of a binary feature to a real-valued target as a p-value. Use the *Kolmogorov-Smirnov* test from from `ks_2samp()` for this.

Parameters

- **x** (*pandas.Series*) – the binary feature vector
- **y** (*pandas.Series*) – the real-valued target vector

Returns the p-value of the feature significance test. Lower p-values indicate a higher feature significance.

Return type float

Raise `ValueError` if the feature is not binary.

`tsfresh.feature_selection.significance_tests.`**`target_real_feature_real_test`**(*x*, *y*)

Calculate the feature significance of a real-valued feature to a real-valued target as a p-value. Use *Kendall's tau* from `kendalltau()` for this.

Parameters

- **x** (*pandas.Series*) – the real-valued feature vector
- **y** (*pandas.Series*) – the real-valued target vector

Returns the p-value of the feature significance test. Lower p-values indicate a higher feature significance.

Return type float

Module contents

The *feature_selection* module contains feature selection algorithms. Those methods were suited to pick the best explaining features out of a massive amount of features. Often the features have to be picked in situations where one has more features than samples. Traditional feature selection methods can be not suitable for such situations which is why we propose a p-value based approach that inspects the significance of the features individually to avoid overfitting and spurious correlations.

tsfresh.scripts package

Submodules

tsfresh.scripts.run_tsfresh module

Run the script with: ``` python run_tsfresh.py path_to_your_csv.csv

Inline literal start-string without end-string.

- Currently this only samples to first 50 values.
- Your csv must be space delimited.
- Output is saved as path_to_your_csv.features.csv

` e.g.: ` python run_tsfresh.py data.txt ```

Inline literal start-string without end-string.

Inline interpreted text or phrase reference start-string without end-string.

A corresponding csv containing time series features will be saved as features_path_to_your_csv.csv `tsfresh.scripts.run_tsfresh.`**`main`**`(`*console_args=None*`)`

Module contents

tsfresh.transformers package

Submodules

tsfresh.transformers.feature_augmenter module

class `tsfresh.transformers.feature_augmenter.`**`FeatureAugmenter`**`(`*default_fc_parameters=None, kind_to_fc_parameters=None, column_id=None, column_sort=None, column_kind=None, column_value=None, timeseries_container=None, parallelization=None, chunksize=None, n_processes=2, show_warnings=False, disable_progressbar=False, impute_function=None, profile=False, profiling_filename='profile.txt', profiling_sorting='cumulative'*`)`

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.TransformerMixin`

Sklearn-compatible estimator, for calculating and adding many features calculated from a given time series to the data. Is is basically a wrapper around `extract_features()`.

The features include basic ones like min, max or median, and advanced features like fourier transformations or statistical tests. For a list of all possible features, see the module *`feature_calculators`*. The column name of each added feature contains the name of the function of that module, which was used for the calculation.

For this estimator, two datasets play a crucial role:

1. the time series container with the timeseries data. This container (for the format see *Data Formats*) contains the data which is used for calculating the features. It must be groupable by ids which are used to identify which feature should be attached to which row in the second dataframe:
2. the input data, where the features will be added to.

Imagine the following situation: You want to classify 10 different financial shares and you have their development in the last year as a time series. You would then start by creating features from the metainformation of the shares, e.g. how long they were on the market etc. and filling up a table - the features of one stock in one row.

```
>>> df = pandas.DataFrame()
>>> # Fill in the information of the stocks
>>> df["started_since_days"] = 0 # add a feature
```

You can then extract all the features from the time development of the shares, by using this estimator:

```
>>> time_series = read_in_timeseries() # get the development of the shares
>>> from tsfresh.transformers import FeatureAugmenter
>>> augmenter = FeatureAugmenter()
>>> augmenter.set_timeseries_container(time_series)
>>> df_with_time_series_features = augmenter.transform(df)
```

The settings for the feature calculation can be controlled with the settings object. If you pass `None`, the default settings are used. Please refer to *`ComprehensiveFCParameters`* for more information.

This estimator does not select the relevant features, but calculates and adds all of them to the DataFrame. See the *`RelevantFeatureAugmenter`* for calculating and selecting features.

For a description what the parameters column_id, column_sort, column_kind and column_value mean, please see *`extraction`*.

`fit`(*X=None*, *y=None*)

The fit function is not needed for this estimator. It just does nothing and is here for compatibility reasons.

Parameters

- **X** (*`Any`*) – Unneeded.
- **y** (*`Any`*) – Unneeded.

Returns The estimator instance itself

Return type *FeatureAugmenter*

`set_timeseries_container`(*timeseries_container*)

Set the timeseries, with which the features will be calculated. For a format of the time series container, please refer to *`extraction`*. The timeseries must contain the same indices as the later DataFrame, to which the features will be added (the one you will pass to *`transform()`*). You can call this function as often as you like, to change the timeseries later (e.g. if you want to extract for different ids).

Parameters **timeseries_container** (*`pandas.DataFrame or dict`*) – The timeseries as a pandas.DataFrame or a dict. See *`extraction`* for the format.

Returns None

Return type None

`transform`(*X*)

Add the features calculated using the timeseries_container and add them to the corresponding rows in the input pandas.DataFrame X.

To save some computing time, you should only include those time serieses in the container, that you need. You can set the timeseries container with the method *`set_timeseries_container()`*.

Parameters **X** (*`pandas.DataFrame`*) – the DataFrame to which the calculated timeseries features will be added. This is *not* the dataframe with the timeseries itself.

Returns The input DataFrame, but with added features.

Return type pandas.DataFrame

tsfresh.transformers.feature_selector module

class `tsfresh.transformers.feature_selector.`**`FeatureSelector`**(*test_for_binary_target_binary_feature='fisher', test_for_binary_target_real_feature='mann', test_for_real_target_binary_feature='mann', test_for_real_target_real_feature='kendall', fdr_level=0.05, hypotheses_independent=False, n_processes=2, chunksize=None*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.TransformerMixin`

Sklearn-compatible estimator, for reducing the number of features in a dataset to only those, that are relevant and significant to a given target. It is basically a wrapper around `check_fs_sig_bh()`.

The check is done by testing the hypothesis

H_0 = the Feature is not relevant and can not be added‘

against

H_1 = the Feature is relevant and should be kept

using several statistical tests (depending on whether the feature or/and the target is binary or not). Using the Benjamini Hochberg procedure, only features in H_0 are rejected.

This estimator - as most of the sklearn estimators - works in a two step procedure. First, it is fitted on training data, where the target is known:

```
>>> import pandas as pd
>>> X_train, y_train = pd.DataFrame(), pd.Series() # fill in with your features␣
↪and target
>>> from tsfresh.transformers import FeatureSelector
>>> selector = FeatureSelector()
>>> selector.fit(X_train, y_train)
```

In this example the list of relevant features is empty: >>> selector.relevant_features >>> []

The same holds for the feature importance: >>> **selector.feature_importances_** >>> array([], dtype=float64)

The estimator keeps track on those features, that were relevant in the training step. If you apply the estimator after the training, it will delete all other features in the testing data sample:

```
>>> X_test = pd.DataFrame()
>>> X_selected = selector.transform(X_test)
```

After that, X_selected will only contain the features that were relevant during the training.

If you are interested in more information on the features, you can look into the member `relevant_features` after the fit.

`fit`(*X, y*)

Extract the information, which of the features are relevent using the given target.

For more information, please see the `check_fs_sig_bh()` function. All columns in the input data sample are treated as feature. The index of all rows in X must be present in y.

Parameters

- **X** (`pandas.DataFrame or numpy.array`) – data sample with the features, which will be classified as relevant or not

- **y** (*pandas.Series or numpy.array*) – target vecotr to be used, to classify the features

Returns the fitted estimator with the information, which features are relevant

Return type *FeatureSelector*

transform(*X*)

Delete all features, which were not relevant in the fit phase.

Parameters X (*pandas.DataSeries or numpy.array*) – data sample with all features, which will be reduced to only those that are relevant

Returns same data sample as X, but with only the relevant features

Return type pandas.DataFrame or numpy.array

tsfresh.transformers.relevant_feature_augmenter module

class `tsfresh.transformers.relevant_feature_augmenter.`**`RelevantFeatureAugmenter`**(*filter_only_tsfresh_fe de- fault_fc_parameters kind_to_fc_paramete col- umn_id=None, col- umn_sort=None, col- umn_kind=None, col- umn_value=None, time- series_container=No par- al- leliza- tion=None, chunk- size=None, im- pute_function=None n_processes=2, show_warnings=Fal dis- able_progressbar=F pro- file=False, pro- fil- ing_filename='profi pro- fil- ing_sorting='cumul test_for_binary_targ test_for_binary_targ test_for_real_target_ test_for_real_target_ fdr_level=0.05, hy- pothe- ses_independent=Fa*

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.TransformerMixin`

Sklearn-compatible estimator to calculate relevant features out of a time series and add them to a data sample.

As many other sklearn estimators, this estimator works in two steps:

In the fit phase, all possible time series features are calculated using the time series, that is set by the set_timeseries_container function (if the features are not manually changed by handing in a feature_extraction_settings object). Then, their significance and relevance to the target is computed using statistical methods and only the relevant ones are selected using the Benjamini Hochberg procedure. These features are stored internally.

In the transform step, the information on which features are relevant from the fit step is used and those features are extracted from the time series. These extracted features are then added to the input data sample.

This estimator is a wrapper around most of the functionality in the tsfresh package. For more information on the subtasks, please refer to the single modules and functions, which are:

- Settings for the feature extraction: `ComprehensiveFCParameters`
- Feature extraction method: `extract_features()`
- Extracted features: `feature_calculators`
- Feature selection: `check_fs_sig_bh()`

This estimator works analogue to the `FeatureAugmenter` with the difference that this estimator does only output and calculate the relevant features, whereas the other outputs all features.

Also for this estimator, two datasets play a crucial role:

1. the time series container with the timeseries data. This container (for the format see `extraction`) contains the data which is used for calculating the features. It must be groupable by ids which are used to identify which feature should be attached to which row in the second dataframe:
2. the input data, where the features will be added to.

Imagine the following situation: You want to classify 10 different financial shares and you have their development in the last year as a time series. You would then start by creating features from the metainformation of the shares, e.g. how long they were on the market etc. and filling up a table - the features of one stock in one row.

```
>>> # Fill in the information of the stocks and the target
>>> X_train, X_test, y_train = pd.DataFrame(), pd.DataFrame(), pd.Series()
```

You can then extract all the relevant features from the time development of the shares, by using this estimator:

```
>>> train_time_series, test_time_series = read_in_timeseries() # get the␣
↪development of the shares
>>> from tsfresh.transformers import RelevantFeatureAugmenter
>>> augmenter = RelevantFeatureAugmenter()
>>> augmenter.set_timeseries_container(train_time_series)
>>> augmenter.fit(X_train, y_train)
>>> augmenter.set_timeseries_container(test_time_series)
>>> X_test_with_features = augmenter.transform(X_test)
```

X_test_with_features will then contain the same information as X_test (with all the meta information you have probably added) plus some relevant time series features calculated on the time series you handed in.

Please keep in mind that the time series you hand in before fit or transform must contain data for the rows that are present in X.

If your set filter_only_tsfresh_features to True, your manually-created features that were present in X_train (or X_test) before using this estimator are not touched. Otherwise, also those features are evaluated and may be rejected from the data sample, because they are irrelevant.

For a description what the parameters column_id, column_sort, column_kind and column_value mean, please see `extraction`.

You can control the feature extraction in the fit step (the feature extraction in the transform step is done automatically) as well as the feature selection in the fit step by handing in settings. However, the default settings which are used if you pass no flags are often quite sensible.

fit (*X*, *y*)

Use the given timeseries from `set_timeseries_container()` and calculate features from it and add them to the data sample X (which can contain other manually-designed features).

Then determine which of the features of X are relevant for the given target y. Store those relevant features internally to only extract them in the transform step.

If filter_only_tsfresh_features is True, only reject newly, automatically added features. If it is False, also look at the features that are already present in the DataFrame.

Parameters

- **`X`** (*`pandas.DataFrame or numpy.array`*) – The data frame without the time series features. The index rows should be present in the timeseries and in the target vector.
- **`y`** (*`pandas.Series or numpy.array`*) – The target vector to define, which features are relevant.

Returns the fitted estimator with the information, which features are relevant.

Return type *RelevantFeatureAugmenter*

`set_timeseries_container`(*timeseries_container*)

Set the timeseries, with which the features will be calculated. For a format of the time series container, please refer to `extraction`. The timeseries must contain the same indices as the later DataFrame, to which the features will be added (the one you will pass to `transform()` or `fit()`). You can call this function as often as you like, to change the timeseries later (e.g. if you want to extract for different ids).

Parameters **`timeseries_container`** (*`pandas.DataFrame or dict`*) – The timeseries as a pandas.DataFrame or a dict. See `extraction` for the format.

Returns None

Return type None

`transform`(*X*)

After the fit step, it is known which features are relevant, Only extract those from the time series handed in with the function `set_timeseries_container()`.

If filter_only_tsfresh_features is False, also delete the irrelevant, already present features in the data frame.

Parameters **`X`** (*`pandas.DataFrame or numpy.array`*) – the data sample to add the relevant (and delete the irrelevant) features to.

Returns a data sample with the same information as X, but with added relevant time series features and deleted irrelevant information (only if filter_only_tsfresh_features is False).

Return type pandas.DataFrame

Module contents

The module `transformers` contains several transformers which can be used inside a sklearn pipeline.

tsfresh.utilities package

Submodules

tsfresh.utilities.dataframe_functions module

Utility functions for handling the DataFrame conversions to the internal normalized format (see `normalize_input_to_internal_representation`) or on how to handle `NaN` and `inf` in the DataFrames.

`tsfresh.utilities.dataframe_functions.`**`check_for_nans_in_columns`**(*df*, *columns=None*)

Helper function to check for `NaN` in the data frame and raise a `ValueError` if there is one.

Parameters

- **`df`** (*`pandas.DataFrame`*) – the pandas DataFrame to test for NaNs
- **`columns`** (*`list`*) – a list of columns to test for NaNs. If left empty, all columns of the DataFrame will be tested.

Returns None

Return type None

Raise `ValueError` of `NaNs` are found in the DataFrame.

`tsfresh.utilities.dataframe_functions.`**`get_range_values_per_column`**(*df*)

Retrieves the finite max, min and mean values per column in the DataFrame *df* and stores them in three dictionaries. Those dictionaries *col_to_max*, *col_to_min*, *col_to_median* map the columnname to the maximal, minimal or median value of that column.

If a column does not contain any finite values at all, a 0 is stored instead.

Parameters **`df`** (*`pandas.DataFrame`*) – the Dataframe to get columnswise max, min and median from

Returns Dictionaries mapping column names to max, min, mean values

Return type (dict, dict, dict)

`tsfresh.utilities.dataframe_functions.`**`impute`**(*df_impute*)

Columnwise replaces all `NaNs` and `infs` from the DataFrame *df_impute* with average/extreme values from the same columns. This is done as follows: Each occurring `inf` or `NaN` in *df_impute* is replaced by

- `-inf` -> `min`
- `+inf` -> `max`
- `NaN` -> `median`

If the column does not contain finite values at all, it is filled with zeros.

This function modifies *df_impute* in place. After that, df_impute is guaranteed to not contain any non-finite values. Also, all columns will be guaranteed to be of type `np.float64`.

Parameters **`df_impute`** (*`pandas.DataFrame`*) – DataFrame to impute

Return df_impute imputed DataFrame

Rtype df_impute pandas.DataFrame

`tsfresh.utilities.dataframe_functions.`**`impute_dataframe_range`**(*df_impute*, *col_to_max*, *col_to_min*, *col_to_median*)

Columnwise replaces all `NaNs`, `-inf` and `+inf` from the DataFrame *df_impute* with average/extreme values from the provided dictionaries.

This is done as follows: Each occurring `inf` or `NaN` in *df_impute* is replaced by

- `-inf` -> by value in col_to_min
- `+inf` -> by value in col_to_max
- `NaN` -> by value in col_to_median

If a column of df_impute is not found in the one of the dictionaries, this method will raise a ValueError. Also, if one of the values to replace is not finite a ValueError is returned

This function modifies *df_impute* in place. Afterwards df_impute is guaranteed to not contain any non-finite values. Also, all columns will be guaranteed to be of type `np.float64`.

Parameters

- **`df_impute`** (*`pandas.DataFrame`*) – DataFrame to impute
- **`col_to_max`** (*`dict`*) – Dictionary mapping column names to max values
- **`col_to_min`** – Dictionary mapping column names to min values
- **`col_to_median`** – Dictionary mapping column names to median values

Return df_impute imputed DataFrame

Rtype df_impute pandas.DataFrame

Raises `ValueError` – if a column of df_impute is missing in col_to_max, col_to_min or col_to_median or a value to replace is non finite

`tsfresh.utilities.dataframe_functions.`**`impute_dataframe_zero`**(*df_impute*)

Replaces all `NaNs`, `-infs` and `+infs` from the DataFrame *df_impute* with 0s. The *df_impute* will be modified in place. All its columns will be into converted into dtype `np.float64`.

Parameters `df_impute` (*`pandas.DataFrame`*) – DataFrame to impute

Return df_impute imputed DataFrame

Rtype df_impute pandas.DataFrame

`tsfresh.utilities.dataframe_functions.`**`normalize_input_to_internal_representation`**(*df_or_dict*, *column_id*, *column_sort*, *column_kind*, *column_value*)

Try to transform any given input to the internal representation of time series, which is a mapping from string (the kind) to a pandas DataFrame with exactly two columns (the value and the id).

This function can transform pandas DataFrames in different formats or dictionaries to pandas DataFrames in different formats. It is used internally in the extract_features function and should not be called by the user.

Parameters

- **`df_or_dict`** (*`pandas.DataFrame or dict`*) – a pandas DataFrame or a dictionary. The required shape/form of the object depends on the rest of the passed arguments.
- **`column_id`** (*`basestring or None`*) – it must be present in the pandas DataFrame or in all DataFrames in the dictionary. It is not allowed to have NaN values in this column.
- **`column_sort`** (*`basestring or None`*) – if not None, sort the rows by this column. It is not allowed to have NaN values in this column.
- **`column_kind`** (*`basestring or None`*) – It can only be used when passing a pandas DataFrame (the dictionary is already assumed to be grouped by the kind). Is must be present in the DataFrame and no NaN values are allowed. The DataFrame will be grouped by the values in the kind column and each group will be one entry in the resulting mapping. If the kind column is not passed, it is assumed that each column in the pandas DataFrame

(except the id or sort column) is a possible kind and the DataFrame is split up into as many DataFrames as there are columns. Except when a value column is given: then it is assumed that there is only one column.

- **column_value** (*basestring or None*) – If it is given, it must be present and not-NaN on the pandas DataFrames (or all pandas DataFrames in the dictionaries). If it is None, it is assumed that there is only a single remaining column in the DataFrame(s) (otherwise an exception is raised).

Returns A tuple of 3 elements: the normalized DataFrame as a dictionary mapping from the kind (as a string) to the corresponding DataFrame, the name of the id column and the name of the value column

Return type (dict, basestring, basestring)

Raise `ValueError` when the passed combination of parameters is wrong or does not fit to the input DataFrame or dict.

`tsfresh.utilities.dataframe_functions.`**`restrict_input_to_index`**(*df_or_dict*, *column_id*, *index*)

Restrict df_or_dict to those ids contained in index.

Parameters

- **df_or_dict** (*pandas.DataFrame or dict*) – a pandas DataFrame or a dictionary.
- **column_id** (*basestring*) – it must be present in the pandas DataFrame or in all DataFrames in the dictionary. It is not allowed to have NaN values in this column.
- **index** (*Iterable or pandas.Series*) – Index containing the ids

Return df_or_dict_restricted the restricted df_or_dict

Rtype df_or_dict_restricted dict or pandas.DataFrame

Raise `TypeError` if df_or_dict is not of type dict or pandas.DataFrame

`tsfresh.utilities.dataframe_functions.`**`roll_time_series`**(*df_or_dict*, *column_id*, *column_sort*, *column_kind*, *rolling_direction*, *maximum_number_of_timeshifts=None*)

Roll the (sorted) data frames for each kind and each id separately in the “time” domain (which is represented by the sort order of the sort column given by *column_sort*).

For each rolling step, a new id is created by the scheme “id={id}, shift={shift}”, here id is the former id of the column and shift is the amount of “time” shifts.

A few remarks:

•This method will create new IDs!

•The sign of rolling defines the direction of time rolling, a positive value means we are going back in time

•It is possible to shift time series of different lenghts but

•We assume that the time series are uniformly sampled

•For more information, please see *How to handle rolling time series*.

Parameters

- **df_or_dict** (*pandas.DataFrame or dict*) – a pandas DataFrame or a dictionary. The required shape/form of the object depends on the rest of the passed arguments.

- **column_id** (*basestring or None*) – it must be present in the pandas DataFrame or in all DataFrames in the dictionary. It is not allowed to have NaN values in this column.
- **column_sort** (*basestring or None*) – if not None, sort the rows by this column. It is not allowed to have NaN values in this column.
- **column_kind** (*basestring or None*) – It can only be used when passing a pandas DataFrame (the dictionary is already assumed to be grouped by the kind). Is must be present in the DataFrame and no NaN values are allowed. If the kind column is not passed, it is assumed that each column in the pandas DataFrame (except the id or sort column) is a possible kind.
- **rolling_direction** (*int*) – The sign decides, if to roll backwards or forwards in "time"
- **maximum_number_of_timeshifts** (*int*) – If not None, shift only up to maximum_number_of_timeshifts. If None, shift as often as possible.

Returns The rolled data frame or dictionary of data frames

Return type the one from df_or_dict

tsfresh.utilities.profiling module

Contains methods to start and stop the profiler that checks the runtime of the different feature calculators

tsfresh.utilities.profiling.**end_profiling**(*profiler*, *filename*, *sorting=None*)

Helper function to stop the profiling process and write out the profiled data into the given filename. Before this, sort the stats by the passed sorting.

Parameters

- **profiler** (*cProfile.Profile*) – An already started profiler (probably by start_profiling).
- **filename** (*basestring*) – The name of the output file to save the profile.
- **sorting** (*basestring*) – The sorting of the statistics passed to the sort_stats function.

Returns None

Return type None

Start and stop the profiler with:

```
>>> profiler = start_profiling()
>>> # Do something you want to profile
>>> end_profiling(profiler, "out.txt", "cumulative")
```

tsfresh.utilities.profiling.**start_profiling**()

Helper function to start the profiling process and return the profiler (to close it later).

Returns a started profiler.

Return type cProfile.Profile

Start and stop the profiler with:

```
>>> profiler = start_profiling()
>>> # Do something you want to profile
>>> end_profiling(profiler, "cumulative", "out.txt")
```

Module contents

This *`utilities`* submodule contains several utility functions. Those should only be used internally inside tsfresh.

Submodules

tsfresh.defaults module

Module contents

At the top level we export the three most important submodules of tsfresh, which are:

- `extract_features`
- `select_features`
- `extract_relevant_features`

Data Formats

tsfresh offers three different options to specify the time series data to be used in the `tsfresh.extract_features()` function (and all utility functions that expect a time series, e.g. the *`tsfresh.utilities.dataframe_functions.roll_time_series()`* function).

Irrespective of the input format, tsfresh will always return the calculated features in the same output format described below.

All three input format options consist of `pandas.DataFrame` objects. There are four important column types that make up those DataFrames. Each will be described with an example from the robot failures dataset (see *Quick Start*).

Mandatory:

column_id This column indicates which entities the time series belong to. Features will be extracted individually for each entity. The resulting feature matrix will contain one row per entity. Each robot is a different entity, so each of it has a different id.

column_value This column contains the actual values of the time series. This corresponds to the measured values for different the sensors on the robots.

Optional (but strongly recommended to specify if you have this column):

column_sort This column contains values which allow to sort the time series (e.g. time stamps). It is not required to have equidistant time steps or the same time scale for the different ids and/or kinds. If you omit this column, the DataFrame is assumed to be already sorted in increasing order. The robot sensor measurements each have a time stamp which is used in this column.

Please note that none of the algorithms of tsfresh uses the actual values in this time column - but only their sorting order.

Optional:

column_kind This column indicates the names of the different time series types (E.g. different sensors in an industrial application as in the robot dataset). For each kind of time series the features are calculated individually.

Important: None of these columns is allowed to contain any `NaN`, `Inf` or `-Inf` values.

In the following we describe the different input formats, that are build on those columns:

- A flat DataFrame
- A stacked DataFrame
- A dictionary of flat DataFrames

The difference between a flat and a stacked DataFrame is indicated by specifying or not specifying the parameters *column_value* and *column_kind* in the `tsfresh.extract_features()` function.

If you do not know which one to choose, you probably want to try out the flat or stacked DataFrame.

Input Option 1. Flat DataFrame

If both *column_value* and *column_kind* are set to `None`, the time series data is assumed to be in a flat DataFrame. This means that each different time series must be saved as its own column.

Example: Imagine you record the values of time series x and y for different objects A and B for three different times t1, t2 and t3. Now you want to calculate some feature with tsfresh. Your resulting DataFrame may look like this:

id	time	x	y
A	t1	x(A, t1)	y(A, t1)
A	t2	x(A, t2)	y(A, t2)
A	t3	x(A, t3)	y(A, t3)
B	t1	x(B, t1)	y(B, t1)
B	t2	x(B, t2)	y(B, t2)
B	t3	x(B, t3)	y(B, t3)

and you would pass

```
column_id="id", column_sort="time", column_kind=None, column_value=None
```

to the extraction functions, to extract features separately for all ids and separately for the x and y values.

Input Option 2. Stacked DataFrame

If both *column_value* and *column_kind* are set, the time series data is assumed to be a stacked DataFrame. This means that there are no different columns for the different types of time series. This representation has several advantages over the flat Data Frame. For example, the time stamps of the different time series do not have to align.

It does not contain different columns for the different types of time series but only one value column and a kind column. The example from above would look like this:

id	time	kind	value
A	t1	x	x(A, t1)
A	t2	x	x(A, t2)
A	t3	x	x(A, t3)
A	t1	y	y(A, t1)
A	t2	y	y(A, t2)
A	t3	y	y(A, t3)
B	t1	x	x(B, t1)
B	t2	x	x(B, t2)
B	t3	x	x(B, t3)
B	t1	y	y(B, t1)
B	t2	y	y(B, t2)
B	t3	y	y(B, t3)

Then you would set

```
column_id="id", column_sort="time", column_kind="kind", column_value="value"
```

to end up with the same extracted features as above.

Input Option 3. Dictionary of flat DataFrames

Instead of passing a DataFrame which must be split up by its different kinds by tsfresh, you can also give a dictionary mapping from the kind as string to a DataFrame containing only the time series data of that kind. So essentially you are using a singular DataFrame for each kind of time series.

The data from the example can be split into two DataFrames resulting in the following dictionary

{ “x”:

id	time	value
A	t1	x(A, t1)
A	t2	x(A, t2)
A	t3	x(A, t3)
B	t1	x(B, t1)
B	t2	x(B, t2)
B	t3	x(B, t3)

, “y”:

id	time	value
A	t1	y(A, t1)
A	t2	y(A, t2)
A	t3	y(A, t3)
B	t1	y(B, t1)
B	t2	y(B, t2)
B	t3	y(B, t3)

}

You would pass this dictionary to tsfresh together with the following arguments:

```
column_id="id", column_sort="time", column_kind=None, column_value="value":
```

In this case we do not need to specify the kind column as the kind is the respective dictionary key.

Output Format

The resulting feature matrix for all three input options will be the same. It will always be a `pandas.DataFrame` with the following layout

id	x_feature_1	...	x_feature_N	y_feature_1	...	y_feature_N
A	...	...	...	...	...	...
B	...	...	...	...	...	...

where the x features are calculated using all x values (independently for A and B), y features using all y values and so on.

This form of DataFrame is also the expected input format to the feature selection algorithms (e.g. the `tsfresh.select_features()` function).

scikit-learn Transformers

tsfresh includes three scikit-learn compatible transformers. You can easily add them to your existing data science pipeline. If you are not familiar with scikit-learn's pipeline we recommend you take a look at the official documentation[1].

The purpose of such a pipeline is to assemble several preprocessing steps that can be cross-validated together while setting different parameters. Our tsfresh transformer allows you to extract and filter the time series features during such a preprocessing sequence.

The first two estimators contained in tsfresh are the `FeatureAugmenter`, which extracts the features, and the `FeatureSelector`, which only performs the feature selection algorithm. It is preferable to combine extracting and filtering of the features in a single step to avoid unnecessary feature calculations. Hence, we have the `RelevantFeatureAugmenter`, which combines both the extraction and filtering of the features in a single step.

Example

In the following example you see how we combine tsfresh's `RelevantFeatureAugmenter` and a `RandomForestClassifier` into a single pipeline. This pipeline can then fit both our transformer and the classifier in one step.

```
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
from tsfresh.examples import load_robot_execution_failures
from tsfresh.transformers import RelevantFeatureAugmenter

pipeline = Pipeline([('augmenter', RelevantFeatureAugmenter(column_id='id', column_
↪sort='time')),
            ('classifier', RandomForestClassifier())])

df_ts, y = load_robot_execution_failures()
X = pd.DataFrame(index=y.index)

pipeline.set_params(augmenter__timeseries_container=df_ts)
pipeline.fit(X, y)
```

The parameters of the augment transformer correspond to the parameters of the top-level convenience function `extract_relevant_features()`. In the example, we only set the names of two columns `column_id='id'`, `column_sort='time'` (see *Data Formats* for an explanation of those parameters).

Because we cannot pass the time series container directly as a parameter to the augmenter step when calling fit or transform on a `sklearn.pipeline.Pipeline` we have to set it manually by calling `pipeline.set_params(augmenter__timeseries_container=df_ts)`. In general, you can change the time series container from which the features are extracted by calling either the pipeline's `set_params()` method or the transformers `set_timeseries_container()` method.

For further examples, see the Jupyter Notebook pipeline_example.ipynb in the notebooks folder of the tsfresh package.

References

[1] http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

Feature Calculation

Overview on extracted feature

tsfresh already calculates a comprehensive number of features. If you are interested in which features are calculated, just go to our

tsfresh.feature_extraction.feature_calculators

module. You will find the documentation of every calculated feature there.

Feature naming

tsfresh enforces a strict naming of the created features, which you have to follow whenever you create new feature calculators. This is due to the *tsfresh.feature_extraction.settings.from_columns()* method which needs to deduce the following information from the feature name

- the time series that was used to calculate the feature
- the feature calculator method that was used to derive the feature
- all parameters that have been used to calculate the feature (optional)

Hence, to enable the *tsfresh.feature_extraction.settings.from_columns()* to deduce all the necessary conditions, the features will be named in the following format

> {time_series_name}__{feature_name}__{parameter name 1}_{parameter value 1}__[..]__{parameter name k}_{parameter value k}

(Here we assumed that {feature_name} has k parameters).

Examples for feature naming

So for example the following feature name

> temperature_1__quantile__q_0.6

is the value of the feature *tsfresh.feature_extraction.feature_calculators.quantile()* for the time series `` `temperature_1` `` and a parameter value of `q=0.6`. On the other hand, the feature named

> Pressure 5__cwt_coefficients__widths_(2, 5, 10, 20)__coeff_14__w_5

denotes the value of the feature *tsfresh.feature_extraction.feature_calculators.cwt_coefficients()* for the time series `` `Pressure 5` `` under parameter values of `widths=(2, 5, 10, 20)`, `coeff=14` and `w=5`.

Feature extraction settings

When starting a new data science project involving time series you probably want to start by extracting a comprehensive set of features. Later you can identify which features are relevant for the task at hand. In the final stages, you probably want to fine tune the parameter of the features to fine tune your models.

You can do all those things with tsfresh. So, you need to know how to control which features are calculated by tsfresh and how one can adjust the parameters. In this section, we will clarify this.

For the lazy: Just let me calculate some features

So, to just calculate a comprehensive set of features, call the `tsfresh.extract_features()` method without passing a *default_fc_parameters* or *kind_to_fc_parameters* object, which means you are using the default options (which will use all feature calculators in this package for what we think are sane default parameters).

For the advanced: How does I set the parameters for all kind of time series?

After digging deeper into your data, you maybe want to calculate more of a certain type of feature and less of another type. So, you need to use custom settings for the feature extractors. To do that with tsfresh you will have to use a custom settings object:

```
>>> from tsfresh.feature_extraction import ComprehensiveFCParameters
>>> settings = ComprehensiveFCParameters()
>>> # Set here the options of the settings object as shown in the paragraphs below
>>> # ...
>>> from tsfresh.feature_extraction import extract_features
>>> extract_features(df, default_fc_parameters=settings)
```

The *default_fc_parameters* is expected to be a dictionary, which maps feature calculator names (the function names you can find in the *`tsfresh.feature_extraction.feature_calculators`* file) to a list of dictionaries, which are the parameters with which the function will be called (as key value pairs). Each function parameter combination, that is in this dict will be called during the extraction and will produce a feature. If the function does not take any parameters, the value should be set to *None*.

For example

```
fc_parameters = {
    "length": None,
    "large_standard_deviation": [{"r": 0.05}, {"r": 0.1}]
}
```

will produce three features: one by calling the *`tsfresh.feature_extraction.feature_calculators.length()`* function without any parameters and two by calling *`tsfresh.feature_extraction.feature_calculators.large_standard_deviation()`* with *r = 0.05* and *r = 0.1*.

So you can control, which features will be extracted, by adding/removing either keys or parameters from this dict. It is as easy as that. If you decide to not calculate the length feature here, you delete it from the dictionary:

```
del fc_parameters["length"]
```

And now, only the two other features are calculated.

For convenience, three dictionaries are predefined and can be used right away:

- *`tsfresh.feature_extraction.settings.ComprehensiveFCParameters`*: includes all features without parameters and all features with parameters, each with different parameter combinations. This is the default for *extract_features* if you do not hand in a *default_fc_parameters* at all.
- *`tsfresh.feature_extraction.settings.MinimalFCParameters`*: includes only a handful of features and can be used for quick tests. The features which have the "minimal" attribute are used here.
- *`tsfresh.feature_extraction.settings.EfficientFCParameters`*: Mostly the same features as in the *`tsfresh.feature_extraction.settings.ComprehensiveFCParameters`*, but without features which are marked with the "high_comp_cost" attribute. This can be used if runtime performance plays a major role.

Theoretically, you could calculate an unlimited number of features with tsfresh by adding entry after entry to the dictionary.

For the ambitious: How do I set the parameters for different type of time series?

It is also possible, to control the features to be extracted for the different kinds of time series individually. You can do so by passing another dictionary to the extract function as a

kind_to_fc_parameters = {"kind" : *fc_parameters*}

parameter. This dict must be a mapping from kind names (as string) to *fc_parameters* objects, which you would normally pass as an argument to the *default_fc_parameters* parameter.

So, for example using

```
kind_to_fc_parameters = {
    "temperature": {"mean": None},
    "pressure": {"max": None, "min": None}
}
```

will extract the *"mean"* feature of the *"temperature"* time series and the *"min"* and *"max"* of the *"pressure"* time series.

The *kind_to_fc_parameters* argument will partly override the *default_fc_parameters*. So, if you include a kind name in the *kind_to_fc_parameters* parameter, its value will be used for that kind. Other kinds will still use the *default_fc_parameters*.

A handy trick: Do I really have to create the dictionary by hand?

Not necessarily. let's assume you have a DataFrame of tsfresh features. By using feature selection algorithms you find out that only a subgroup of features is relevant.

Then, we provide the `tsfresh.feature_extraction.settings.from_columns()` method that constructs the *kind_to_fc_parameters* dictionary from the column names of this filtered feature matrix to make sure that only relevant features are extracted.

This can save a huge amount of time because you prevent the calculation of uncessary features. Let's illustrate that with an example:

```
# X_tsfresh containes the extracted tsfresh features
X_tsfresh = extract_features(...)

# which are now filtered to only contain relevant features
X_tsfresh_filtered = some_feature_selection(X_tsfresh, y, ....)

# we can easily construct the corresponding settings object
kind_to_fc_parameters = tsfresh.settings.from_columns(X_tsfresh_filtered)
```

this will construct you the *kind_to_fc_parameters* dictionary that corresponds to the features and parameters (!) from the tsfresh features that were filtered by the *some_feature_selection* feature selection algorithm.

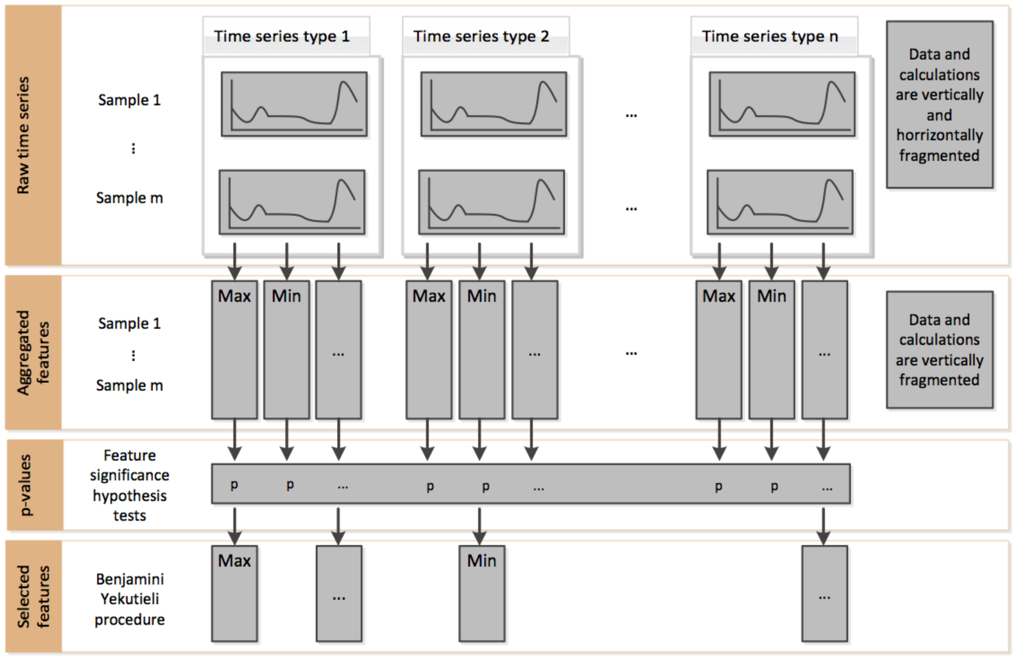
Feature filtering

The all-relevant problem of feature selection is the identification of all strongly and weakly relevant attributes. This problem is especially hard to solve for time series classification and regression in industrial applications such as predictive maintenance or production line optimization, for which each label or regression target is associated with several time series and meta-information simultaneously.

To limit the number of irrelevant features, tsfresh deploys the fresh algorithm (fresh stands for *FeatuRe Extraction based on Scalable Hypothesis tests*)[1].

The algorithm is called by *tsfresh.feature_selection.feature_selector.check_fs_sig_bh()*. It is an efficient, scalable feature extraction algorithm, which filters the available features in an early stage of the machine learning pipeline with respect to their significance for the classification or regression task, while controlling the expected percentage of selected but irrelevant features.

The filtering process consists of three phases which are sketched in the following figure:


Raw time series
Sample 1
Sample m
Time series type 1
Time series type 2
Time series type n
Data and calculations are vertically and horrizontally fragmented
Aggregated features
Max
Min
Data and calculations are vertically fragmented
p-values
Feature significance hypothesis tests
p
Selected features
Benjamini Yekutieli procedure

Phase 1 - Feature extraction

Firstly, the algorithm characterizes time series with comprehensive and well-established feature mappings and considers additional features describing meta-information. The feature calculators used to derive the features are contained in *tsfresh.feature_extraction.feature_calculators*.

In the figure from above, this corresponds to the change from raw time series to aggregated features.

Phase 2 - Feature significance testing

In a second step, each feature vector is individually and independently evaluated with respect to its significance for predicting the target under investigation. Those tests are contained in the submodule *tsfresh.feature_selection.significance_tests*. The result of these tests is a vector of p-values, quantifying the significance of each feature for predicting the label/target.

In the figure from above, this corresponds to the change from aggregated features to p-values.

[1] Christ, M., Kempa-Liehr, A.W. and Feindt, M. (2016). Distributed and parallel time series feature extraction for industrial big data applications. ArXiv e-prints: 1610.07717 URL: http://adsabs.harvard.edu/abs/2016arXiv161007717C

Phase 3 - Multiple test procedure

The vector of p-values is evaluated on basis of the Benjamini-Yekutieli procedure[2] in order to decide which features to keep. This multiple testing procedure is contained in the submodule `tsfresh.feature_selection.feature_selector`.

In the figure from above, this corresponds to the change from p-values to selected features.

References

How to add a custom feature

It may be beneficial to add a custom feature to those that are calculated by tsfresh. To do so, one has to adapt certain steps:

Step 1. Decide which type of feature you want to implement

In tsfresh we differentiate between three types of feature calculation methods

1. aggregate features without parameter
2. aggregate features with parameter
3. apply features with parameters

So if you want to add a singular feature with out any parameters, stick with *1.*, the aggregate feature without parameters.

Then, if your features can be calculated independently for each possible parameter set, stick with type *2.*, the aggregate features with parameters.

If both cases from above do not apply, because it is beneficial to calculate the features for the different parameter settings at the same time (to e.g. perform auxiliary calculations only once for all features), stick with type *3.*, the apply features with parameters.

Step 2. Write the feature calculator

Depending on which type of feature you are implementing, you can use the following feature calculator skeletons:

1. aggregate features without parameter

```
@set_property("fctype", "aggregate")
def your_feature_calculator(x):
    """
    The description of your feature

    :param x: the time series to calculate the feature of
    :type x: pandas.Series
    :return: the value of this feature
    :return type: bool or float
    """
```

[2] Benjamini, Y. and Yekutieli, D. (2001). The control of the false discovery rate in multiple testing under dependency. Annals of statistics, 1165–1188

```
    # Calculation of feature as float, int or bool
    f = f(x)
    return f
```

2. aggregate features with parameter

```
@set_property("fctype", "aggregate_with_parameters")
def your_feature_calculator(x, p1, p2, ...):
    """
    Description of your feature

    :param x: the time series to calculate the feature of
    :type x: pandas.Series
    :param p1: description of your parameter p1
    :type p1: type of your parameter p1
    :param p2: description of your parameter p2
    :type p2: type of your parameter p2
    ...
    :return: the value of this feature
    :return type: bool or float
    """
    # Calculation of feature as float, int or bool
    f = f(x)
    return f
```

3. apply features with parameters

```
@set_property("fctype", "apply")
def your_feature_calculator(x, c, param):
    """
    Description of your feature

    :param x: the time series to calculate the feature of
    :type x: pandas.Series
    :param c: the time series name
    :type c: str
    :param param: contains dictionaries {"p1": x, "p2": y, ...} with p1 float, p2 int␣
↪...
    :type param: list
    :return: the different feature values
    :return type: pandas.Series
    """
    # Calculation of feature as pandas.Series s, the index is the name of the feature
    s = f(x)
    return s
```

After implementing the feature calculator, please add it to the *tsfresh.feature_extraction.feature_calculators* submodule. tsfresh will only find feature calculators that are in this submodule.

Step 3. Add custom settings for your feature

Finally, you have to add custom settings if your feature is an apply or aggregate feature with parameters. To do so, just append your feature with sane default parameters to the `name_to_param` dictionary inside the `tsfresh.ComprehensiveFCParameters` constructor:

```
name_to_param.update({
    # here are the existing settings
    ...
    # Now the settings of your feature calculator
    "your_feature_calculator" = [{"p1": x, "p2": y, ...} for x,y in ...],
})
```

That is it, tsfresh will calculate your feature the next time you run it.

Please make sure, that the different feature extraction settings (e.g. `tsfresh.feature_extraction.settings.EfficientCParameters`, *tsfresh.feature_extraction.settings.MinimalFCParameters* or *tsfresh.feature_extraction.settings.ComprehensiveFCParameters*) do include different sets of feature calculators to use. You can control, which feature extraction settings object will include your new feature calculator by giving your function attributes like "minimal" or "high_comp_cost". Please see the classes in *tsfresh.feature_extraction.settings* for more information.

Step 4. Add a pull request

We would very happy if you contribute your implemented features to tsfresh. So make sure to create a pull request at our github page. We happily accept partly implemented features that we can finalize.

Parallelization

The feature extraction as well as the feature selection offer the possibility of parallelization. Out of the box both tasks are parallelized by tsfresh. However, the overhead introduced with the parallelization should not be underestimated. Here we discuss the different settings to control the parallelization. To achieve best results for your use-case you should experiment with the parameters.

Please let us know about your results tuning the below mentioned parameters! It will help improve this document as well as the default settings.

Parallelization of Feature Selection

We use a `multiprocessing.Pool` to parallelize the calculation of the p-values for each feature. On instantiation we set the Pool's number of worker processes to *n_processes*. This field defaults to the number of processors on the current system. We recommend setting it to the maximum number of available (and otherwise idle) processors.

The chunksize of the Pool's map function is another important parameter to consider. It can be set via the *chunksize* field. By default it is up to `multiprocessing.Pool` to decide on the chunksize.

Parallelization of Feature Extraction

For the feature extraction tsfresh exposes the parameters *n_processes* and *chunksize*. Both behave anlogue to the parameters for the feature selection.

Additionally there are two options for how the parallelization is done:

1. `'per_kind'` parallelizes the feature calculation per kind of time series.
2. `'per_sample'` parallelizes per kind and per sample.

To enforce an option, either pass `'per_kind'` or `'per_sample'` as the `parallelization=` parameter of the `tsfresh.extract_features()` function. By default the option is chosen with a rule of thumb:

If the number of different time series (kinds) is less than half of the number of available worker processes (`n_processes`) then `'per_sample'` is chosen, otherwise `'per_kind'`.

Generally, there is no perfect setting for all cases. On the one hand more parallelization can speed up the calculation as the work is better distributed among the computers resources. On the other hand parallelization introduces overheads such as copying data to the worker processes, splitting the data to enable the distribution and combining the results.

Implementing the parallelization we observed the following aspects:

- For small data sets the difference between parallelization per kind or per sample should be negligible.
- For data sets with one kind of time series parallelization per sample results in a decent speed up that grows with the number of samples.
- The more kinds of time series the data set contains, the more samples are necessary to make parallelization per sample worthwhile.
- If the data set contains more kinds of time series than available cpu cores, parallelization per kind is the way to go.

How to handle rolling time series

Lets assume that we have a DataFrame of one of the tsfresh *Data Formats*. The “sort” column of such a container gives a sequential state to the individual measurements. In the case of time series this can be the *time* dimension while in the case of spectra the order is given by the *wavelength* or *frequency* dimensions. We can exploit this sequence to generate more input data out of single time series, by *rolling* over the data.

Imagine the following situation: You have the data of certain sensors (e.g. EEG measurements) as the base to classify patients into a healthy and not healthy group (we oversimplify the problem here). Lets say you have sensor data of 100 time steps, so you may extract features for the forecasting of the patients healthiness by a classification algorithm. If you also have measurements of the healthiness for those 100 time steps (this is the target vector), then you could predict the healthiness of the patient in every time step, which essentially states a time series forecasting problem. So, to do that, you want to extract features in every time step of the original time series while for example looking at the last 10 steps. A rolling mechanism creates such time series for every time step by creating sub time series of the sensor data of the last 10 time steps.

Another example can be found in streaming data, e.g. in Industry 4.0 applications. Here you typically get one new data row at a time and use this to for example predict machine failures. To train your model, you could act as if you would stream the data, by feeding your classifier the data after one time step, the data after the first two time steps etc.

Both examples imply, that you extract the features not only on the full data set, but also on all temporal coherent subsets of data, which is the process of *rolling*. In tsfresh, this is implemented in the function *`tsfresh.utilities.dataframe_functions.roll_time_series()`*.

The rolling mechanism takes a time series x with its data rows $[x_1, x_2, x_3, ..., x_n]$ and creates n new time series $\hat{x}^k$, each of them with a different consecutive part of x:

$$\hat{x}^k = [x_k, x_{k-1}, x_{k-2}, ..., x_1]$$

To see what this does in real-world applications, we look into the following example flat DataFrame in tsfresh format

id	time	x	y
1	t1	1	5
1	t2	2	6
1	t3	3	7
1	t4	4	8
2	t8	10	12
2	t9	11	13

where you have measured the values from two sensors x and y for two different entities (id 1 and 2) in 4 or 2 time steps (t1 to t9).

Now, we can use `tsfresh.utilities.dataframe_functions.roll_time_series()` to get consecutive sub-time series. E.g. if you set *rolling* to 0, the feature extraction works on the original time series without any rolling.

So it extracts 2 set of features,

id	time	x	y
1	t1	1	5
1	t2	2	6
1	t3	3	7
1	t4	4	8

and

id	time	x	y
2	t8	10	12
2	t9	11	13

If you set rolling to 1, the feature extraction works with all of the following time series:

id	time	x	y
1	t1	1	5

id	time	x	y
1	t1	1	5
1	t2	2	6

id	time	x	y
1	t1	1	5
1	t2	2	6
1	t3	3	7
2	t8	10	12

id	time	x	y
1	t1	1	5
1	t2	2	6
1	t3	3	7
1	t4	4	8
2	t8	10	12
2	t9	11	13

If you set rolling to -1, you end up with features for the time series, rolled in the other direction

id	time	x	y
1	t4	4	8

id	time	x	y
1	t3	3	7
1	t4	4	8

id	time	x	y
1	t2	2	6
1	t3	3	7
1	t4	4	8
2	t9	11	13

id	time	x	y
1	t1	1	5
1	t2	2	6
1	t3	3	7
1	t4	4	8
2	t8	10	12
2	t9	11	13

We only gave an example for the flat DataFrame format, but rolling actually works on all 3 *Data Formats* that are supported by tsfresh.

FAQ

1. **Does tsfresh support different time series lengths?**

   Yes, it supports different time series lengths. However, some feature calculators can demand a minimal length of the time series. If a shorter time series is passed to the calculator, a NaN is returned for those features.

2. **Is it possible to extract features from rolling/shifted time series?**

   Yes, the `tsfresh.dataframe_functions.roll_time_series()` function allows to conviniently create a rolled time series datframe from your data. You just have to transform your data into one of the supported tsfresh *Data Formats*. Then, the `tsfresh.dataframe_functions.roll_time_series()` give you a DataFrame with the rolled time series, that you can pass to tsfresh. On the following page you can find a detailed description: *How to handle rolling time series*.

3. **How can I use tsfresh with windows?**

   We recommend to use Anaconda. After installing, open the Anaconda Prompt, create an environment and set up tsfresh (Please be aware that we're using multiprocessing, which can be problematic.):

```
conda create -n ENV_NAME python=VERSION
conda install -n ENV_NAME pip requests numpy pandas scipy statsmodels patsy␣
↪scikit-learn future six tqdm
activate ENV_NAME
pip install tsfresh
```

4. **Does tsfresh support different sampling rates in the time series?**

   Yes! The feature calculators in tsfresh do not care about the sampling frequency. You will have to use the second input format, the stacked DataFramed (see *Data Formats*)

Authors

Development Lead

- Maximilian Christ (maximilianchrist.com, max.christ@me.com)

Contributions

- Nils Braun (nilslennartbraun@gmail.com)
- Julius Neuffer (julius.neuffer@blue-yonder.com)

License

Changelog

tsfresh uses Semantic Versioning

Version 0.8.0

- **Breaking API changes:**
  - removing of feature extraction settings object, replaced by keyword arguments and a plain dictionary (fc_parameters)
  - removing of feature selection settings object, replaced by keyword arguments
- added notebook with examples of new API
- added chapter in docs about the new API
- adjusted old notebooks and documentation to new API

Version 0.7.1

- added a maximum shift parameter to the rolling utility
- added a FAQ entry about how to use tsfresh on windows
- **drastically decreased the runtime of the following features**
    - cwt_coefficient
    - index_mass_quantile
    - number_peaks
    - large_standard_deviation
    - symmetry_looking
- removed baseline unit tests
- **bugfixes:**
    - per sample parallel imputing was done on chunks which gave non deterministic results
    - imputing on dtypes other that float32 did not work properly
- several improvements to documentation

Version 0.7.0

- new rolling utility to use tsfresh for time series forecasting tasks
- **bugfixes:**
    - index_mass_quantile was using global index of time series container
    - an index with same name as id_column was breaking parallelization
    - friedrich_coefficients and max_langevin_fixed_point were occasionally stalling

Version 0.6.0

- progress bar for feature selection
- new feature: estimation of largest fixed point of deterministic dynamics
- new notebook: demonstration how to use tsfresh in a pipeline with train and test datasets
- remove no logging handler warning
- fixed bug in the RelevantFeatureAugmenter regarding the evaluate_only_added_features parameters

Version 0.5.0

- new example: driftbif simulation
- further improvements of the parallelization
- language improvements in the documentation
- performance improvements for some features
- performance improvements for the impute function

- new feature and feature renaming: sum_of_recurring_values, sum_of_recurring_data_points

Version 0.4.0

- fixed several bugs: checking of UCI dataset, out of index error for mean_abs_change_quantiles
- added a progress bar denoting the progress of the extraction process
- added parallelization per sample
- added unit tests for comparing results of feature extraction to older snapshots
- added “high_comp_cost” attribute
- added ReasonableFeatureExtraction settings only calculating features without “high_comp_cost” attribute

Version 0.3.1

- fixed several bugs: closing multiprocessing pools / index out of range cwt calculator / division by 0 in index_mass_quantile
- now all warnings are disabled by default
- for a singular type time series data, the name of value column is used as feature prefix

Version 0.3.0

- fixed bug with parsing of “NUMBER_OF_CPUS” environment variable
- now features are calculated in parallel for each type

Version 0.2.0

- now p-values are calculated in parallel
- fixed bugs for constant features
- allow time series columns to be named 0
- moved uci repository datasets to github mirror
- added feature calculator sample_entropy
- added MinimalFeatureExtraction settings
- fixed bug in calculation of fourier coefficients

Version 0.1.2

- added support for python 3.5.2
- fixed bug with the naming of the features that made the naming of features non-deterministic

Version 0.1.1

- mainly fixes for the read-the-docs documentation, the pypi readme and so on

Version 0.1.0

- Initial version :)

How to contribute

We want tsfresh to become the biggest archive of feature extraction methods in python. To achieve this goal, we need your help!

All contributions, bug reports, bug fixes, documentation improvements, enhancements and ideas are welcome. If you want to add one or two interesting feature calculators, implement a new feature selection process or just fix 1-2 typos, your help is appreciated.

If you want to help, just create a pull request on our github page. To the new user, working with Git can sometimes be confusing and frustrating. If you are not familiar with Git you can also contact us by *email*.

Guidelines

There are three general coding paradigms that we believe in:

1. **Keep it simple**. We believe that *"Programs should be written for people to read, and only incidentally for machines to execute."*.
2. **Keep it documented** by at least including a docstring for each method and class. Do not describe what you are doing but why you are doing it.
3. **Keep it tested**. We aim for a high test coverage.

There are two important copyright guidelines:

4. Please do not include any data sets for which a licence is not available or commercial use is even prohibited. Those can undermine the licence of the whole projects.
5. Do not use code snippets for which a licence is not available (e.g. from stackoverflow) or commercial use is even prohibited. Those can undermine the licence of the whole projects.

Further, there are some technical decisions we made:

6. Clear the Output of iPython notebooks. This improves the readability of related Git diffs.

Test framework

After making your changes, you probably want to test your changes locally. To run our comprehensive suite of unit tests you have to install all the relevant python packages with

```
cd /path/to/tsfresh
pip install -r requirements.txt
pip install -r rdocs-requirements.txt
pip install -r test-requirements.txt
pip install -e .
```

The last command will dynamically link the tsfresh package which means that changes to the code will directly show up for example in your test run.

Then, if you have everything installed, you can run the tests with

```
python setup.py test
```

or build the documentation with

```
python setup.py docs
```

The finished documentation can be found in the docs/_build/html folder.

On Github we use a Travis CI Folder that runs our test suite every time a commit or pull request is sent. The configuration of Travi is controlled by the .travis.yml file.

We are looking forward to hear from you! =)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Acknowledgements

The research and development of TSFRESH was funded in part by the German Federal Ministry of Education and Research under grant number 01IS14004 (project iPRODICT).

Python Module Index

Index