tslearn Documentation

Release 0.4.0

Romain Tavenard

Dec 07, 2020

Contents

`tslearn` is a Python package that provides machine learning tools for the analysis of time series. This package builds on (and hence depends on) `scikit-learn`, `numpy` and `scipy` libraries.

This documentation contains *a quick-start guide* (including *installation procedure* and *basic usage of the toolkit*), *a complete API Reference*, as well as a *gallery of examples*.

Finally, if you use `tslearn` in a scientific publication, *we would appreciate citations*.

CHAPTER 1

Quick-start guide

For a list of functions and classes available in `tslearn`, please have a look at our *API Reference*.

1.1 Installation

1.1.1 Using conda

The easiest way to install `tslearn` is probably via `conda`:

```
conda install -c conda-forge tslearn
```

1.1.2 Using PyPI

Using `pip` should also work fine:

```
python -m pip install tslearn
```

In this case, you should have `numpy`, `cython` and C++ build tools available at build time.

1.1.3 Using latest github-hosted version

If you want to get `tslearn`'s latest version, you can refer to the repository hosted at github:

```
python -m pip install https://github.com/tslearn-team/tslearn/archive/master.zip
```

In this case, you should have `numpy`, `cython` and C++ build tools available at build time.

It seems on some platforms `Cython` dependency does not install properly. If you experiment such an issue, try installing it with the following command:

```
python -m pip install cython
```

before you start installing `tslearn`. If it still does not work, we suggest you switch to *conda* installation.

1.1.4 Other requirements

`tslearn` builds on (and hence depends on) `scikit-learn`, `numpy` and `scipy` libraries.

If you plan to use the *`tslearn.shapelets`* module from `tslearn`, `tensorflow` (v2) should also be installed. `h5py` is required for reading or writing models using the hdf5 file format. In order to load multivariate datasets from the UCR/UEA archive using the *`tslearn.datasets.UCR_UEA_datasets`* class, installed `scipy` version should be greater than 1.3.0.

1.2 Getting started

This tutorial will guide you to format your first time series data, import standard datasets, and manipulate them using dedicated machine learning algorithms.

1.2.1 Time series format

First, let us have a look at what `tslearn` time series format is. To do so, we will use the `to_time_series` utility from *`tslearn.utils`*:

```
>>> from tslearn.utils import to_time_series
>>> my_first_time_series = [1, 3, 4, 2]
>>> formatted_time_series = to_time_series(my_first_time_series)
>>> print(formatted_time_series.shape)
(4, 1)
```

In `tslearn`, a time series is nothing more than a two-dimensional `numpy` array with its first dimension corresponding to the time axis and the second one being the feature dimensionality (1 by default).

Then, if we want to manipulate sets of time series, we can cast them to three-dimensional arrays, using `to_time_series_dataset`. If time series from the set are not equal-sized, NaN values are appended to the shorter ones and the shape of the resulting array is `(n_ts, max_sz, d)` where `max_sz` is the maximum of sizes for time series in the set.

```
>>> from tslearn.utils import to_time_series_dataset
>>> my_first_time_series = [1, 3, 4, 2]
>>> my_second_time_series = [1, 2, 4, 2]
>>> formatted_dataset = to_time_series_dataset([my_first_time_series, my_second_time_
↪series])
>>> print(formatted_dataset.shape)
(2, 4, 1)
>>> my_third_time_series = [1, 2, 4, 2, 2]
>>> formatted_dataset = to_time_series_dataset([my_first_time_series,
                                                my_second_time_series,
                                                my_third_time_series])
>>> print(formatted_dataset.shape)
(3, 5, 1)
```

1.2.2 Importing standard time series datasets

If you aim at experimenting with standard time series datasets, you should have a look at the *tslearn.datasets*.

```
>>> from tslearn.datasets import UCR_UEA_datasets
>>> X_train, y_train, X_test, y_test = UCR_UEA_datasets().load_dataset("TwoPatterns")
>>> print(X_train.shape)
(1000, 128, 1)
>>> print(y_train.shape)
(1000,)
```

Note that when working with time series datasets, it can be useful to rescale time series using tools from the *tslearn.preprocessing*.

If you want to import other time series from text files, the expected format is:

- each line represents a single time series (and time series from a dataset are not forced to be the same length);
- in each line, modalities are separated by a | character (useless if you only have one modality in your data);
- in each modality, observations are separated by a space character.

Here is an example of such a file storing two time series of dimension 2 (the first time series is of length 3 and the second one is of length 2).

```
1.0 0.0 2.5|3.0 2.0 1.0
1.0 2.0|4.333 2.12
```

To read from / write to this format, have a look at the *tslearn.utils*:

```
>>> from tslearn.utils import save_time_series_txt, load_time_series_txt
>>> time_series_dataset = load_time_series_txt("path/to/your/file.txt")
>>> save_time_series_txt("path/to/another/file.txt", dataset_to_be_saved)
```

1.2.3 Playing with your data

Once your data is loaded and formatted according to `tslearn` standards, the next step is to feed machine learning models with it. Most `tslearn` models inherit from `scikit-learn` base classes, hence interacting with them is very similar to interacting with a `scikit-learn` model, except that datasets are not two-dimensional arrays, but rather `tslearn` time series datasets (*i.e.* three-dimensional arrays or lists of two-dimensional arrays).

```
>>> from tslearn.clustering import TimeSeriesKMeans
>>> km = TimeSeriesKMeans(n_clusters=3, metric="dtw")
>>> km.fit(X_train)
```

As seen above, one key parameter when applying machine learning methods to time series datasets is the metric to be used. You can learn more about it in the *dedicated section* of this documentation.

1.3 Methods for variable-length time series

This page lists machine learning methods in *tslearn* that are able to deal with datasets containing time series of different lengths. We also provide example usage for these methods using the following variable-length time series dataset:

```
from tslearn.utils import to_time_series_dataset
X = to_time_series_dataset([[1, 2, 3, 4], [1, 2, 3], [2, 5, 6, 7, 8, 9]])
y = [0, 0, 1]
```

1.3.1 Classification

- *tslearn.neighbors.KNeighborsTimeSeriesClassifier*
- *tslearn.svm.TimeSeriesSVC*
- *tslearn.shapelets.LearningShapelets*

Examples

```
from tslearn.neighbors import KNeighborsTimeSeriesClassifier
knn = KNeighborsTimeSeriesClassifier(n_neighbors=2)
knn.fit(X, y)
```

```
from tslearn.svm import TimeSeriesSVC
clf = TimeSeriesSVC(C=1.0, kernel="gak")
clf.fit(X, y)
```

```
from tslearn.shapelets import LearningShapelets
clf = LearningShapelets(n_shapelets_per_size={3: 1})
clf.fit(X, y)
```

1.3.2 Regression

- *tslearn.svm.TimeSeriesSVR*

Examples

```
from tslearn.svm import TimeSeriesSVR
clf = TimeSeriesSVR(C=1.0, kernel="gak")
y_reg = [1.3, 5.2, -12.2]
clf.fit(X, y_reg)
```

1.3.3 Nearest-neighbor search

- *tslearn.neighbors.KNeighborsTimeSeries*

Examples

```
from tslearn.neighbors import KNeighborsTimeSeries
knn = KNeighborsTimeSeries(n_neighbors=2)
knn.fit(X)
knn.kneighbors()    # Search for neighbors using series from `X` as queries
knn.kneighbors(X2)  # Search for neighbors using series from `X2` as queries
```

1.3.4 Clustering

- *tslearn.clustering.KernelKMeans*

- *tslearn.clustering.TimeSeriesKMeans*
- *tslearn.clustering.silhouette_score*

Examples

```
from tslearn.clustering import KernelKMeans
gak_km = KernelKMeans(n_clusters=2, kernel="gak")
labels_gak = gak_km.fit_predict(X)
```

```
from tslearn.clustering import TimeSeriesKMeans
km = TimeSeriesKMeans(n_clusters=2, metric="dtw")
labels = km.fit_predict(X)
km_bis = TimeSeriesKMeans(n_clusters=2, metric="softdtw")
labels_bis = km_bis.fit_predict(X)
```

```
from tslearn.clustering import TimeSeriesKMeans, silhouette_score
km = TimeSeriesKMeans(n_clusters=2, metric="dtw")
labels = km.fit_predict(X)
silhouette_score(X, labels, metric="dtw")
```

1.3.5 Barycenter computation

- *tslearn.barycenters.dtw_barycenter_averaging*
- *tslearn.barycenters.softdtw_barycenter*

Examples

```
from tslearn.barycenters import dtw_barycenter_averaging
bar = dtw_barycenter_averaging(X, barycenter_size=3)
```

```
from tslearn.barycenters import softdtw_barycenter
from tslearn.utils import ts_zeros
initial_barycenter = ts_zeros(sz=5)
bar = softdtw_barycenter(X, init=initial_barycenter)
```

1.3.6 Model selection

Also, model selection tools offered by `scikit-learn` can be used on variable-length data, in a standard way, such as:

```
from sklearn.model_selection import KFold, GridSearchCV
from tslearn.neighbors import KNeighborsTimeSeriesClassifier

knn = KNeighborsTimeSeriesClassifier(metric="dtw")
p_grid = {"n_neighbors": [1, 5]}

cv = KFold(n_splits=2, shuffle=True, random_state=0)
clf = GridSearchCV(estimator=knn, param_grid=p_grid, cv=cv)
clf.fit(X, y)
```

1.3.7 Resampling

- *tslearn.preprocessing.TimeSeriesResampler*

Finally, if you want to use a method that cannot run on variable-length time series, one option would be to first resample your data so that all your time series have the same length and then run your method on this resampled version of your dataset.

Note however that resampling will introduce temporal distortions in your data. Use with great care!

```
from tslearn.preprocessing import TimeSeriesResampler

resampled_X = TimeSeriesResampler(sz=X.shape[1]).fit_transform(X)
```

1.4 Integration with other Python packages

`tslearn` is a general-purpose Python machine learning library for time series that offers tools for pre-processing and feature extraction as well as dedicated models for clustering, classification and regression. To ensure compatibility with more specific Python packages, we provide utilities to convert data sets from and to other formats.

tslearn.utils.to_time_series_dataset() is a general function that transforms an array-like object into a three-dimensional array of shape `(n_ts, sz, d)` with the following conventions:

- the fist axis is the sample axis, `n_ts` being the number of time series;
- the second axis is the time axis, `sz` being the maximum number of time points;
- the third axis is the dimension axis, `d` being the number of dimensions.

This is how a data set of time series is represented in `tslearn`.

The following sections briefly explain how to transform a data set from `tslearn` to another supported Python package and vice versa.

1.4.1 scikit-learn

scikit-learn is a popular Python package for machine learning. *tslearn.utils.to_sklearn_dataset()* converts a data set from `tslearn` format to `scikit-learn` format. To convert a data set from `scikit-learn`, you can use *tslearn.utils.to_time_series_dataset()*.

```
>>> from tslearn.utils import to_sklearn_dataset
>>> to_sklearn_dataset([[1, 2], [1, 4, 3]])
array([[ 1.,  2., nan],
       [ 1.,  4.,  3.]])
>>> to_time_series_dataset([[ 1.,  2., None], [ 1.,  4.,  3.]])
array([[[ 1.],
    [ 2.],
    [nan]],

   [[ 1.],
    [ 4.],
    [ 3.]]])
```

1.4.2 pyts

pyts is a Python package dedicated to time series classification. *tslearn.utils.to_pyts_dataset()* and *tslearn.utils.from_pyts_dataset()* allow users to convert a data set from `tslearn` format to `pyts` format and vice versa.

```
>>> from tslearn.utils import from_pyts_dataset, to_pyts_dataset
>>> from_pyts_dataset([[1, 2], [1, 4]])
array([[[1],
        [2]],

       [[1],
        [4]]])

>>> to_pyts_dataset([[[1], [2]], [[1], [4]]])
array([[1., 2.],
       [1., 4.]])
```

1.4.3 seglearn

seglearn is a python package for machine learning time series or sequences. *tslearn.utils.to_seglearn_dataset()* and *tslearn.utils.from_seglearn_dataset()* allow users to convert a data set from `tslearn` format to `seglearn` format and vice versa.

```
>>> from tslearn.utils import from_seglearn_dataset, to_seglearn_dataset
>>> from_seglearn_dataset([[1, 2], [1, 4, 3]])
array([[[ 1.],
        [ 2.],
        [nan]],

       [[ 1.],
        [ 4.],
        [ 3.]]])
>>> to_seglearn_dataset([[[1], [2], [None]], [[1], [4], [3]]])
array([array([[1.],
       [2.]]),
       array([[1.],
       [4.],
       [3.]])], dtype=object)
```

1.4.4 stumpy

stumpy is a powerful and scalable Python library for computing a Matrix Profile, which can be used for a variety of time series data mining tasks. *tslearn.utils.to_stumpy_dataset()* and *tslearn.utils.from_stumpy_dataset()* allow users to convert a data set from `tslearn` format to `stumpy` format and vice versa.

```
>>> import numpy as np
>>> from tslearn.utils import from_stumpy_dataset, to_stumpy_dataset
>>> from_stumpy_dataset([np.array([1, 2]), np.array([1, 4, 3])])
array([[[ 1.],
        [ 2.],
        [nan]],
```

(continues on next page)

(continued from previous page)

```
       [[ 1.],
        [ 4.],
        [ 3.]]])
>>> to_stumpy_dataset([[[1], [2], [None]], [[1], [4], [3]]])
[array([1., 2.]), array([1., 4., 3.])]
```

1.4.5 sktime

sktime is a `scikit-learn` compatible Python toolbox for learning with time series. *`tslearn.utils.to_sktime_dataset()`* and *`tslearn.utils.from_sktime_dataset()`* allow users to convert a data set from `tslearn` format to `sktime` format and vice versa. `pandas` is a required dependency to use these functions.

```
>>> import pandas as pd
>>> from tslearn.utils import from_sktime_dataset, to_sktime_dataset
>>> df = pd.DataFrame()
>>> df["dim_0"] = [pd.Series([1, 2]), pd.Series([1, 4, 3])]
>>> from_sktime_dataset(df)
array([[[ 1.],
        [ 2.],
        [nan]],

       [[ 1.],
        [ 4.],
        [ 3.]]])
>>> to_sktime_dataset([[[1], [2], [None]], [[1], [4], [3]]]).shape
(2, 1)
```

1.4.6 pyflux

pyflux is a library for time series analysis and prediction. *`tslearn.utils.to_pyflux_dataset()`* and *`tslearn.utils.from_pyflux_dataset()`* allow users to convert a data set from `tslearn` format to `pyflux` format and vice versa. `pandas` is a required dependency to use these functions.

```
>>> import pandas as pd
>>> from tslearn.utils import from_pyflux_dataset, to_pyflux_dataset
>>> df = pd.DataFrame([1, 2], columns=["dim_0"])
>>> from_pyflux_dataset(df)
array([[[1.],
        [2.]]])
>>> to_pyflux_dataset([[[1], [2]]]).shape
(2, 1)
```

1.4.7 tsfresh

tsfresh iis a python package automatically calculating a large number of time series characteristics. *`tslearn.utils.to_tsfresh_dataset()`* and *`tslearn.utils.from_tsfresh_dataset()`* allow users to convert a data set from `tslearn` format to `tsfresh` format and vice versa. `pandas` is a required dependency to use these functions.

```
>>> import pandas as pd
>>> from tslearn.utils import from_tsfresh_dataset, to_tsfresh_dataset
```

(continues on next page)

(continued from previous page)

```
>>> df = pd.DataFrame([[0, 0, 1.0],
...                    [0, 1, 2.0],
...                    [1, 0, 1.0],
...                    [1, 1, 4.0],
...                    [1, 2, 3.0]], columns=['id', 'time', 'dim_0'])
>>> from_tsfresh_dataset(df)
array([[[ 1.],
        [ 2.],
        [nan]],

       [[ 1.],
        [ 4.],
        [ 3.]]])
>>> to_tsfresh_dataset([[[1], [2], [None]], [[1], [4], [3]]]).shape
(5, 3)
```

1.4.8 cesium

cesium is an open-source platform for time series inference. *tslearn.utils.to_cesium_dataset()* and *tslearn.utils.from_cesium_dataset()* allow users to convert a data set from `tslearn` format to `cesium` format and vice versa. `cesium` is a required dependency to use these functions.

```
>>> from tslearn.utils import from_cesium_dataset, to_cesium_dataset
>>> from cesium.data_management import TimeSeries
>>> from_cesium_dataset([TimeSeries(m=[1, 2]), TimeSeries(m=[1, 4, 3])])
array([[[ 1.],
        [ 2.],
        [nan]],

       [[ 1.],
        [ 4.],
        [ 3.]]])
>>> len(to_cesium_dataset([[[1], [2], [None]], [[1], [4], [3]]]))
2
```

1.5 Contributing

First of all, thank you for considering contributing to `tslearn`. It's people like you that will help make `tslearn` a great toolkit.

Contributions are managed through GitHub Issues and Pull Requests.

We are welcoming contributions in the following forms:

- **Bug reports**: when filing an issue to report a bug, please use the search tool to ensure the bug hasn't been reported yet;
- **New feature suggestions**: if you think `tslearn` should include a new algorithm, please open an issue to ask for it (of course, you should always check that the feature has not been asked for yet :). Think about linking to a pdf version of the paper that first proposed the method when suggesting a new algorithm.
- **Bug fixes and new feature implementations**: if you feel you can fix a reported bug/implement a suggested feature yourself, do not hesitate to:
    1. fork the project;

2. implement your bug fix;
3. submit a pull request referencing the ID of the issue in which the bug was reported / the feature was suggested;

If you would like to contribute by implementing a new feature reported in the Issues, maybe starting with Issues that are attached the "good first issue" label would be a good idea.

When submitting code, please think about code quality, adding proper docstrings including doctests with high code coverage.

1.5.1 More details on Pull requests

The preferred workflow for contributing to tslearn is to fork the main repository on GitHub, clone, and develop on a branch. Steps:

1. Fork the project repository by clicking on the 'Fork' button near the top right of the page. This creates a copy of the code under your GitHub user account. For more details on how to fork a repository see this guide.
2. Clone your fork of the tslearn repo from your GitHub account to your local disk:

```
$ git clone git@github.com:YourLogin/tslearn.git
$ cd tslearn
```

3. Create a `my-feature` branch to hold your development changes. Always use a `my-feature` branch. It's good practice to never work on the `master` branch:

```
$ git checkout -b my-feature
```

4. Develop the feature on your feature branch. To record your changes in git, add changed files using `git add` and then `git commit` files:

```
$ git add modified_files
$ git commit
```

5. Push the changes to your GitHub account with:

```
$ git push -u origin my-feature
```

6. Follow these instructions to create a pull request from your fork. This will send an email to the committers.

(If any of the above seems like magic to you, please look up the Git documentation on the web, or ask a friend or another contributor for help.)

Pull Request Checklist

We recommended that your contribution complies with the following rules before you submit a pull request:

- Follow the PEP8 Guidelines.
- If your pull request addresses an issue, please use the pull request title to describe the issue and mention the issue number in the pull request description. This will make sure a link back to the original issue is created.
- All public methods should have informative docstrings with sample usage presented as doctests when appropriate.
- Please prefix the title of your pull request with `[MRG]` (Ready for Merge), if the contribution is complete and ready for a detailed review. An incomplete contribution – where you expect to do more work before receiving a full review – should be prefixed `[WIP]` (to indicate a work in progress) and changed to `[MRG]` when it matures.

WIPs may be useful to: indicate you are working on something to avoid duplicated work, request broad review of functionality or API, or seek collaborators. WIPs often benefit from the inclusion of a task list in the PR description.

- When adding additional functionality, provide at least one example script in the `tslearn/docs/examples/` folder. Have a look at other examples for reference. Examples should demonstrate why the new functionality is useful in practice and, if possible, compare it to other methods available in tslearn.
- Documentation and high-coverage tests are necessary for enhancements to be accepted. Bug-fixes or new features should be provided with non-regression tests. These tests verify the correct behavior of the fix or feature. In this manner, further modifications on the code base are granted to be consistent with the desired behavior. For the Bug-fixes case, at the time of the PR, this tests should fail for the code base in master and pass for the PR code.
- At least one paragraph of narrative documentation with links to references in the literature (with PDF links when possible) and the example.

Here is a description of useful tools to check your code locally:

- No PEP8 or PEP257 errors; check with the flake8 Python package:

```
$ pip install flake8
$ flake8 path/to/module.py  # check for errors in one file
$ flake8 path/to/folder  # check for errors in all the files in a folder
$ git diff -u | flake8 --diff  # check for errors in the modified code only
```

- To run the tests locally and get code coverage, use the pytest and pytest-cov Python packages:

```
$ pip install pytest pytest-cov
$ pytest --cov tslearn
```

- To build the documentation locally, install the following packages and run the `make html` command in the `tslearn/docs` folder:

```
$ pip install sphinx==1.8.5 sphinx-gallery sphinx-bootstrap-theme nbsphinx
$ pip install numpydoc matplotlib
$ cd tslearn/docs
$ make html
```

The documentation will be generated in the `_build/html`. You can double click on `index.html` to open the index page, which will look like the first page that you see on the online documentation. Then you can move to the pages that you modified and have a look at your changes.

Bonus points for contributions that include a performance analysis with a benchmark script and profiling output.

CHAPTER 2

User Guide

2.1 Dynamic Time Warping

Dynamic Time Warping (DTW)[1] is a similarity measure between time series. Let us consider two time series $x = (x_0, \ldots, x_{n-1})$ and $y = (y_0, \ldots, y_{m-1})$ of respective lengths n and m. Here, all elements x_i and y_j are assumed to lie in the same d-dimensional space. In `tslearn`, such time series would be represented as arrays of respective shapes *(n, d)* and *(m, d)* and DTW can be computed using the following code:

```
from tslearn.metrics import dtw, dtw_path

dtw_score = dtw(x, y)
# Or, if the path is also an important information:
optimal_path, dtw_score = dtw_path(x, y)
```

2.1.1 Optimization problem

DTW between x and y is formulated as the following optimization problem:

$$DTW(x, y) = \min_{\pi} \sqrt{\sum_{(i,j)\in\pi} d(x_i, y_j)^2}$$

where $\pi = [\pi_0, \ldots, \pi_K]$ is a path that satisfies the following properties:

- it is a list of index pairs $\pi_k = (i_k, j_k)$ with $0 \leq i_k < n$ and $0 \leq j_k < m$
- $\pi_0 = (0, 0)$ and $\pi_K = (n - 1, m - 1)$
- for all $k > 0$, $\pi_k = (i_k, j_k)$ is related to $\pi_{k-1} = (i_{k-1}, j_{k-1})$ as follows:
  - $i_{k-1} \leq i_k \leq i_{k-1} + 1$
  - $j_{k-1} \leq j_k \leq j_{k-1} + 1$

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

Here, a path can be seen as a temporal alignment of time series such that Euclidean distance between aligned (ie. resampled) time series is minimal.

The following image exhibits the DTW path (in white) for a given pair of time series, on top of the cross-similarity matrix that stores $d(x_i, y_j)$ values.

Code to produce such visualization is available in our Gallery of examples.

2.1.2 Algorithmic solution

There exists an $O(mn)$ algorithm to compute the exact optimum for this problem (pseudo-code is provided for time series indexed from 1 for simplicity):

```
def dtw(x, y):
    # Initialization
    for i = 1..n
        for j = 1..m
            C[i, j] = inf

    C[0, 0] = 0.

   # Main loop
   for i = 1..n
        for j = 1..m
            dist = d(x_i, y_j) ** 2
            C[i, j] = dist + min(C[i-1, j], C[i, j-1], C[i-1, j-1])

   return sqrt(C[n, m])
```

2.1.3 Using a different ground metric

By default, `tslearn` uses squared Euclidean distance as the base metric (i.e. $d(\cdot, \cdot)$ in the optimization problem above is the Euclidean distance). If one wants to use another ground metric, the code would then be:

```
from tslearn.metrics import dtw_path_from_metric
path, cost = dtw_path_from_metric(x, y, metric=compatible_metric)
```

in which case the optimization problem that would be solved would be:

$$DTW(x, y) = \min_{\pi} \sum_{(i,j)\in\pi} \tilde{d}(x_i, y_j)$$

where $\tilde{d}(\cdot, \cdot)$ is the user-defined ground metric, denoted `compatible_metric` in the code snippet above.

2.1.4 Properties

Dynamic Time Warping holds the following properties:

- $\forall x, y, DTW(x, y) \geq 0$
- $\forall x, DTW(x, x) = 0$

However, mathematically speaking, DTW is not a valid distance since it does not satisfy the triangular inequality.

2.1.5 Additional constraints

The set of temporal deformations to which DTW is invariant can be reduced by setting additional constraints on the set of acceptable paths. These constraints typically consists in forcing paths to lie close to the diagonal.

First, the Sakoe-Chiba band is parametrized by a radius r (number of off-diagonal elements to consider, also called warping window size sometimes), as illustrated below:

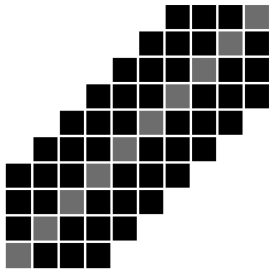

Fig. 1: $n = m = 10, r = 3$. Diagonal is marked in grey for better readability.

The corresponding code would be:

```
from tslearn.metrics import dtw
cost = dtw(x, y, global_constraint="sakoe_chiba", sakoe_chiba_radius=3)
```

Second, the Itakura parallelogram sets a maximum slope s for alignment paths, which leads to a parallelogram-shaped constraint:

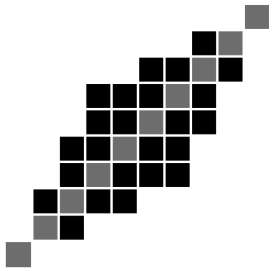

Fig. 2: $n = m = 10, s = 2$. Diagonal is marked in grey for better readability.

The corresponding code would be:

```
from tslearn.metrics import dtw
cost = dtw(x, y, global_constraint="itakura", itakura_max_slope=2.)
```

Alternatively, one can put an upper bound on the warping path length so as to discard complex paths, as described in[2]:

```
from tslearn.metrics import dtw_limited_warping_length
cost = dtw_limited_warping_length(x, y, max_length)
```

[2] Z. Zhang, R. Tavenard, A. Bailly, X. Tang, P. Tang, T. Corpetti Dynamic time warping under limited warping path length. Information Sciences, vol. 393, pp. 91–107, 2017.

2.1.6 Barycenters

Computing barycenter (also known as Fréchet means) of a set $\mathcal{D}$ for DTW corresponds to the following optimization problem:

$$\min_{\mu} \sum_{x \in \mathcal{D}} DTW(\mu, x)^2$$

Optimizing this quantity can be done through the DTW Barycenter Averaging (DBA) algorithm presented in[3].

```
from tslearn.barycenters import dtw_barycenter_averaging
b = dtw_barycenter_averaging(dataset)
```

This is the algorithm at stake when invoking *tslearn.clustering.TimeSeriesKMeans* with `metric="dtw"`.

2.1.7 soft-DTW

DTW is not differentiable with respect to its inputs because of the non-differentiability of the `min` operation. A differentiable extension has been presented in[4] in which the `min` operator is replaced by `soft-min`, using the log-sum-exp formulation:

$$\text{soft-min}_\gamma(a_1, \dots, a_n) = -\gamma \log \sum_i e^{-a_i/\gamma}$$

soft-DTW hence depends on a hyper-parameter γ that controls the smoothing of the resulting metric (squared DTW corresponds to the limit case $\gamma \to 0$).

```
from tslearn.metrics import soft_dtw
soft_dtw_score = soft_dtw(x, y, gamma=.1)
```

Also, barycenters for soft-DTW can be estimated through gradient descent:

```
from tslearn.barycenters import softdtw_barycenter
b = softdtw_barycenter(dataset, gamma=.1)
```

This is the algorithm at stake when invoking *tslearn.clustering.TimeSeriesKMeans* with `metric="softdtw"`.

2.1.8 Examples Involving DTW variants

- *DTW computation*
- *DTW computation with a custom distance metric*
- *Barycenters*
- *Soft-DTW weighted barycenters*

[3] F. Petitjean, A. Ketterlin & P. Gancarski. A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition, Elsevier, 2011, Vol. 44, Num. 3, pp. 678-693

[4] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.

2.1.9 References

2.2 Kernel Methods

In the following, we will discuss the use of kernels to compare time series. A kernel $k(\cdot, \cdot)$ is such that there exists an unknown map Φ such that:

$$k(\mathbf{x}, \mathbf{y}) = \langle \Phi(\mathbf{x}), \Phi(\mathbf{y}) \rangle_{\mathcal{H}}$$

i.e. $k(\cdot, \cdot)$ is the inner product between $\mathbf{x}$ and $\mathbf{y}$ in some (unknown) embedding space $\mathcal{H}$. In practice, $k(\mathbf{x}, \mathbf{y})$ will be large when $\mathbf{x}$ and $\mathbf{y}$ are similar and close to 0 when they are very dissimilar.

A large number of kernel methods from the machine learning literature rely on the so-called *kernel trick*, that consists in performing computations in the embedding space $\mathcal{H}$ without ever actually performing any embedding. As an example, one can compute distance between $\mathbf{x}$ and $\mathbf{y}$ in $\mathcal{H}$ *via*:

$$\begin{aligned}
\|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\|_{\mathcal{H}}^2 &= \langle \Phi(\mathbf{x}) - \Phi(\mathbf{y}), \Phi(\mathbf{x}) - \Phi(\mathbf{y}) \rangle_{\mathcal{H}} \\
&= \langle \Phi(\mathbf{x}), \Phi(\mathbf{x}) \rangle_{\mathcal{H}} + \langle \Phi(\mathbf{y}), \Phi(\mathbf{y}) \rangle_{\mathcal{H}} - 2 \langle \Phi(\mathbf{x}), \Phi(\mathbf{y}) \rangle_{\mathcal{H}} \\
&= k(\mathbf{x}, \mathbf{x}) + k(\mathbf{y}, \mathbf{y}) - 2k(\mathbf{x}, \mathbf{y})
\end{aligned}$$

Such computations are used, for example, in the kernel-k-means algorithm (see below).

2.2.1 Global Alignment Kernel

The Global Alignment Kernel (GAK) is a kernel that operates on time series.

It is defined, for a given bandwidth σ, as:

$$k(\mathbf{x}, \mathbf{y}) = \sum_{\pi \in \mathcal{A}(\mathbf{x}, \mathbf{y})} \prod_{i=1}^{|\pi|} \exp\left(-\frac{\left\| x_{\pi_1(i)} - y_{\pi_2 j} \right\|^2}{2\sigma^2} \right)$$

where $\mathcal{A}(\mathbf{x}, \mathbf{y})$ is the set of all possible alignments between series $\mathbf{x}$ and $\mathbf{y}$.

It is advised in[1] to set the bandwidth σ as a multiple of a simple estimate of the median distance of different points observed in different time-series of your training set, scaled by the square root of the median length of time-series in the set. This estimate is made available in `tslearn` through *tslearn.metrics.sigma_gak*:

```
from tslearn.metrics import gak, sigma_gak

sigma = sigma_gak(X)
k_01 = gak(X[0], X[1], sigma=sigma)
```

Note however that, on long time series, this estimate can lead to numerical overflows, which smaller values can avoid.

Finally, GAK is related to *softDTW*[3] through the following formula:

$$k(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\text{softDTW}_\gamma(\mathbf{x}, \mathbf{y})}{\gamma} \right)$$

where γ is the hyper-parameter controlling softDTw smoothness, which is related to the bandwidth parameter of GAK through $\gamma = 2\sigma^2$.

[1] M. Cuturi. "Fast Global Alignment Kernels," ICML 2011.

[3] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.

2.2.2 Clustering and Classification

Kernel k-means[2] is a method that uses the kernel trick to implicitly perform k-means clustering in the embedding space associated to a kernel. This method is discussed in *our User Guide section dedicated to clustering*.

Kernels can also be used in classification settings. `tslearn.svm` offers implementations of Support Vector Machines (SVM) that accept GAK as a kernel. This implementation heavily relies on `scikit-learn` and `libsvm`. One implication is that `predict_proba` and `predict_log_proba` methods are computed based on cross-validation probability estimates, which has two main implications, as discussed in more details in `scikit-learn`'s user guide:

1. setting the constructor option `probability` to `True` makes the `fit` step longer since it then relies on an expensive five-fold cross-validation;

2. the probability estimates obtained through `predict_proba` may be inconsistent with the scores provided by `decision_function` and the predicted class output by `predict`.

2.2.3 Examples Using Kernel Methods

- *SVM and GAK*
- *Kernel k-means*

2.2.4 References

2.3 Time Series Clustering

Clustering is the task of grouping together similar objects. This task hence heavily relies on the notion of similarity one relies on.

The following Figure illustrates why choosing an adequate similarity function is key (code to reproduce is available *in the Gallery of Examples*).

Fig. 3: k-means clustering with Euclidean distance. Each subfigure represents series from a given cluster and their centroid (in red).

This Figure is the result of a k-means clustering that uses Euclidean distance as a base metric. One issue with this metric is that it is not invariant to time shifts, while the dataset at stake clearly holds such invariants.

2.3.1 k-means and Dynamic Time Warping

To overcome the previously illustrated issue, distance metrics dedicated to time series, such as *Dynamic Time Warping (DTW)*, are required. As can be seen in the Figure below, the use of such metrics produce more meaningful results.

The `tslearn.clustering` module in `tslearn` offers an option to use DTW as the core metric in a k-means algorithm, which leads to better clusters and centroids:

Fig. 4: k-means clustering with Dynamic Time Warping. Each subfigure represents series from a given cluster and their centroid (in red).

[2] I. S. Dhillon, Y. Guan & B. Kulis. "Kernel k-means, Spectral Clustering and Normalized Cuts," KDD 2004.

First, clusters gather time series of similar shapes, which is due to the ability of Dynamic Time Warping (DTW) to deal with time shifts, as explained above. Second, cluster centers (aka centroids) are computed as the barycenters with respect to DTW, hence they allow to retrieve a sensible average shape whatever the temporal shifts in the cluster (see *our dedicated User Guide section* for more details on how these barycenters are computed).

In `tslearn`, clustering a time series dataset with k-means and a dedicated time series metric is as easy as

```
from tslearn.clustering import TimeSeriesKMeans

model = TimeSeriesKMeans(n_clusters=3, metric="dtw",
                         max_iter=10, random_state=seed)
model.fit(X_train)
```

where `X_train` is the considered unlabelled dataset of time series. The `metric` parameter can also be set to `"softdtw"` as an alternative time series metric (*cf. our User Guide section on soft-DTW*).

2.3.2 Kernel k-means and Time Series Kernels

Another option to deal with such time shifts is to rely on the kernel trick. Indeed,[1] introduces a positive semidefinite kernel for time series, inspired from DTW. Then, the kernel k-means algorithm[2], that is equivalent to a k-means that would operate in the Reproducing Kernel Hilbert Space associated to the chosen kernel, can be used:

Fig. 5: Kernel k-means clustering with Global Alignment Kernel. Each subfigure represents series from a given cluster.

A first significant difference (when compared to k-means) is that cluster centers are never computed explicitly, hence time series assignments to cluster are the only kind of information available once the clustering is performed.

Second, one should note that the clusters generated by kernel-k-means are phase dependent (see clusters 2 and 3 that differ in phase rather than in shape). This is because Global Alignment Kernel is not invariant to time shifts, as demonstrated in[3] for the closely related soft-DTW[4].

2.3.3 Examples Using Clustering Estimators

- *k-means*
- *Kernel k-means*

2.3.4 References

2.4 Shapelets

Shapelets are defined in[1] as "subsequences that are in some sense maximally representative of a class". Informally, if we assume a binary classification setting, a shapelet is discriminant if it is **present** in most series of one class and

[1] M. Cuturi. "Fast Global Alignment Kernels," ICML 2011.

[2] I. S. Dhillon, Y. Guan & B. Kulis. "Kernel k-means, Spectral Clustering and Normalized Cuts," KDD 2004.
[3] H. Janati, M. Cuturi, A. Gramfort. "Spatio-Temporal Alignments: Optimal transport through space and time," AISTATS 2020
[4] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.
[1] L. Ye & E. Keogh. Time series shapelets: a new primitive for data mining. SIGKDD 2009.

absent from series of the other class. To assess the level of presence, one uses shapelet matches:

$$d(\mathbf{x}, \mathbf{s}) = \min_t \|\mathbf{x}_{t \to t+L} - \mathbf{s}\|_2$$

where L is the length (number of timestamps) of shapelet $\mathbf{s}$ and $\mathbf{x}_{t \to t+L}$ is the subsequence extracted from time series $\mathbf{x}$ that starts at time index t and stops at $t+L$. If the above-defined distance is small enough, then shapelet $\mathbf{s}$ is supposed to be present in time series $\mathbf{x}$.

Fig. 6: The distance from a time series to a shapelet is done by sliding the shorter shapelet over the longer time series and calculating the point-wise distances. The minimal distance found is returned.

In a classification setting, the goal is then to find the most discriminant shapelets given some labeled time series data. Shapelets can be mined from the training set[1] or learned using gradient-descent.

2.4.1 Learning Time-series Shapelets

`tslearn` provides an implementation of "Learning Time-series Shapelets", introduced in[2], that is an instance of the latter category. In Learning Shapelets, shapelets are learned such that time series represented in their shapelet-transform space (*i.e.* their distances to each of the shapelets) are linearly separable. A shapelet-transform representation of a time series $\mathbf{x}$ given a set of shapelets $\{\mathbf{s}_i\}_{i \leq k}$ is the feature vector: $[d(\mathbf{x}, \mathbf{s}_1), \cdots, d(\mathbf{x}, \mathbf{s}_k)]$. This is illustrated below with a two-dimensional example.

Fig. 7: An example of how time series are transformed into linearly separable distances.

In `tslearn`, in order to learn shapelets and transform timeseries to their corresponding shapelet-transform space, the following code can be used:

```
from tslearn.shapelets import LearningShapelets

model = LearningShapelets(n_shapelets_per_size={3: 2})
model.fit(X_train, y_train)
train_distances = model.transform(X_train)
test_distances = model.transform(X_test)
shapelets = model.shapelets_as_time_series_
```

A *`tslearn.shapelets.LearningShapelets`* model has several hyper-parameters, such as the maximum number of iterations and the batch size. One important hyper-parameters is the `n_shapelets_per_size` which is a dictionary where the keys correspond to the desired lengths of the shapelets and the values to the desired number of shapelets per length. When set to `None`, this dictionary will be determined by a *heuristic*. After creating the model, we can `fit` the optimal shapelets using our training data. After a fitting phase, the distances can be calculated using the `transform` function. Moreover, you can easily access the learned shapelets by using the `shapelets_as_time_series_` attribute.

It is important to note that due to the fact that a technique based on gradient-descent is used to learn the shapelets, our model can be prone to numerical issues (e.g. exploding and vanishing gradients). For that reason, it is important to normalize your data. This can be done before passing the data to the `fit` and `transform` methods, by using our *`tslearn.preprocessing`* module but this can be done internally by the algorithm itself by setting the `scale` parameter.

[2] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

2.4.2 Examples Involving Shapelet-based Estimators

- *Learning Shapelets*
- *Aligning discovered shapelets with timeseries*
- *Learning Shapelets: decision boundaries in 2D distance space*

2.4.3 References

2.5 Matrix Profile

The Matrix Profile, MP, is a new time series that can be calculated based on an input time series T and a subsequence length m. MP_i corresponds to the minimal distance from the query subsequence $T_{i \to i+m}$ to any subsequence in T[1]. As the distance from the query subsequence to itself will be equal to zero, $T_{i-\frac{m}{4} \to i+\frac{m}{4}}$ is considered as an exclusion zone. In order to construct the Matrix Profile, a distance profile which is *similar to the distance calculation used to transform time series into their shapelet-transform space*, is calculated for each subsequence, as illustrated below:

Fig. 8: For each segment, the distances to all subsequences of the time series are calculated and the minimal distance that does not correspond to the original location of the segment (where the distance is zero) is returned.

2.5.1 Implementation

The Matrix Profile implementation provided in `tslearn` uses numpy or wraps around STUMPY[2]. Three different versions are available:

- `numpy`: a slow implementation
- `stump`: a fast CPU version, which requires STUMPY to be installed
- `gpu_stump`: the fastest version, which requires STUMPY to be installed and a GPU

2.5.2 Possible Applications

The Matrix Profile allows for many possible applications, which are well documented on the page created by the original authors[3]. Some of these applications include: motif and shapelet extraction, discord detection, earthquake detection, and many more.

2.5.3 Examples Involving Matrix Profile

- *Matrix Profile*
- *Distance and Matrix Profiles*

[1] C. M. Yeh, Y. Zhu, L. Ulanova, N.Begum et al. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. ICDM 2016.
[2] https://github.com/TDAmeritrade/stumpy
[3] https://www.cs.ucr.edu/~eamonn/MatrixProfile.html

2.5.4 References

2.6 Early Classification of Time Series

Early classification of time series is the task of performing a classification as early as possible for an incoming time series, and decision about when to trigger the decision is part of the prediction process.

2.6.1 Early Classification Cost Function

Dachraoui et al.[1] introduces a composite loss function for early classification of time series that balances earliness and accuracy.

The cost function is of the following form:

$$\mathcal{L}(\mathbf{x}_{\rightarrow t}, y, t, \boldsymbol{\theta}) = \mathcal{L}_c(\mathbf{x}_{\rightarrow t}, y, \boldsymbol{\theta}) + \alpha t$$

where $\mathcal{L}_c(\cdot, \cdot, \cdot)$ is a classification loss and t is the time at which a decision is triggered by the system ($\mathbf{x}_{\rightarrow t}$ is time series $\mathbf{x}$ observed up to time t). In this setting, α drives the tradeoff between accuracy and earliness and is supposed to be a hyper-parameter of the method.

The authors rely on (i) a clustering of the training time series and (ii) individual classifiers $m_t(\cdot)$ trained at all possible timestamps, so as to be able to predict, at time t, an expected cost for all future times $t + \tau$ with $\tau \geq 0$:

$$f_\tau(\mathbf{x}_{\rightarrow t}, y) = \sum_k \left[P(C_k|\mathbf{x}_{\rightarrow t}) \sum_i \left(P(y = i|C_k) \left(\sum_{j \neq i} P_{t+\tau}(\hat{y} = j|y = i, C_k) \right) \right) \right] + \alpha t$$

where:

- $P(C_k|\mathbf{x}_{\rightarrow t})$ is a soft-assignment weight of $\mathbf{x}_{\rightarrow t}$ to cluster C_k;
- $P(y = i|C_k)$ is obtained from a contingency table that stores the number of training time series of each class in each cluster;
- $P_{t+\tau}(\hat{y} = j|y = i, C_k)$ is obtained through training time confusion matrices built on time series from cluster C_k using classifier $m_{t+\tau}(\cdot)$.

At test time, if a series is observed up to time t and if, for all positive τ we have $f_\tau(\mathbf{x}_{\rightarrow t}, y) \geq f_0(\mathbf{x}_{\rightarrow t}, y)$, then a decision is made using classifier $m_t(\cdot)$.

Fig. 9: Early classification. At test time, prediction is made at a timestamp such that the expected earliness-accuracy is optimized, which can hence vary between time series.

To use this early classifier in `tslearn`, one can rely on the *`tslearn.early_classification.NonMyopicEarlyClassifier`* class:

```
from tslearn.early_classification import NonMyopicEarlyClassifier

early_clf = NonMyopicEarlyClassifier(n_clusters=3,
                                     cost_time_parameter=1e-3,
                                     lamb=1e2,
                                     random_state=0)
early_clf.fit(X_train, y_train)
preds, times = early_clf.predict_class_and_earliness(X_test)
```

[1] A. Dachraoui, A. Bondu and A. Cornuejols. "Early classification of time series as a non myopic sequential decision making problem," ECML/PKDD 2015

where `cost_time_parameter` is the α parameter presented above and `lamb` is a trade-off parameter for the soft-assignment of partial series to clusters $P(C_k|\mathbf{x}_{\rightarrow t})$ (when `lamb` tends to infinity, the assignment tends to hard-assignment, and when `lamb` is set to 0, equal probabilities are obtained for all clusters).

2.6.2 Examples Involving Early Classification Estimators

- *Early Classification*

2.6.3 References

CHAPTER 3

API Reference

The complete `tslearn` project is automatically documented for every module.

tslearn.barycenters	The *tslearn.barycenters* module gathers algorithms for time series barycenter computation.
tslearn.clustering	The *tslearn.clustering* module gathers time series specific clustering algorithms.
tslearn.datasets	The *tslearn.datasets* module provides simplified access to standard time series datasets.
tslearn.early_classification	The *tslearn.early_classification* module gathers early classifiers for time series.
tslearn.generators	The *tslearn.generators* module gathers synthetic time series dataset generation routines.
tslearn.matrix_profile	The *tslearn.matrix_profile* module gathers methods for the computation of Matrix Profiles from time series.
tslearn.metrics	The *tslearn.metrics* module delivers time-series specific metrics to be used at the core of machine learning algorithms.
tslearn.neural_network	The *tslearn.neural_network* module contains multi-layer perceptron models for time series classification and regression.
tslearn.neighbors	The *tslearn.neighbors* module gathers nearest neighbor algorithms using time series metrics.
tslearn.piecewise	The *tslearn.piecewise* module gathers time series piecewise approximation algorithms.
tslearn.preprocessing	The *tslearn.preprocessing* module gathers time series scalers and resamplers.
tslearn.shapelets	The *tslearn.shapelets* module gathers Shapelet-based algorithms.

Continued on next page

Table 1 – continued from previous page

tslearn.svm	The tslearn.svm module contains Support Vector Classifier (SVC) and Support Vector Regressor (SVR) models for time series.
tslearn.utils	The tslearn.utils module includes various utilities.

3.1 tslearn.barycenters

The tslearn.barycenters module gathers algorithms for time series barycenter computation.

A barycenter (or *Fréchet mean*) is a time series b which minimizes the sum of squared distances to the time series of a given data set x:

$$\min \sum_i d(b, x_i)^2$$

Only the methods dtw_barycenter_averaging() and softdtw_barycenter() can operate on variable-length time-series (see *here*).

See the *barycenter examples* for an overview.

Functions

euclidean_barycenter(X[, weights])	Standard Euclidean barycenter computed from a set of time series.
dtw_barycenter_averaging(X[, ...])	DTW Barycenter Averaging (DBA) method estimated through Expectation-Maximization algorithm.
dtw_barycenter_averaging_subgradient(X[, ...])	DTW Barycenter Averaging (DBA) method estimated through subgradient descent algorithm.
softdtw_barycenter(X[, gamma, weights, ...])	Compute barycenter (time series averaging) under the soft-DTW [1] geometry.

3.1.1 tslearn.barycenters.euclidean_barycenter

tslearn.barycenters.**euclidean_barycenter**(*X*, *weights=None*)

Standard Euclidean barycenter computed from a set of time series.

Parameters

X [array-like, shape=(n_ts, sz, d)] Time series dataset.

weights: None or array Weights of each X[i]. Must be the same size as len(X). If None, uniform weights are used.

Returns

numpy.array of shape (sz, d) Barycenter of the provided time series dataset.

Notes

This method requires a dataset of equal-sized time series

Examples

```
>>> time_series = [[1, 2, 3, 4], [1, 2, 4, 5]]
>>> bar = euclidean_barycenter(time_series)
>>> bar.shape
(4, 1)
>>> bar
array([[1. ],
       [2. ],
       [3.5],
       [4.5]])
```

Examples using `tslearn.barycenters.euclidean_barycenter`

- *Barycenters*

3.1.2 tslearn.barycenters.dtw_barycenter_averaging

`tslearn.barycenters.`**`dtw_barycenter_averaging`**(*X*, *barycenter_size=None*, *init_barycenter=None*, *max_iter=30*, *tol=1e-05*, *weights=None*, *metric_params=None*, *verbose=False*, *n_init=1*)

DTW Barycenter Averaging (DBA) method estimated through Expectation-Maximization algorithm.

DBA was originally presented in [1]. This implementation is based on a idea from [2] (Majorize-Minimize Mean Algorithm).

Parameters

X [array-like, shape=(n_ts, sz, d)] Time series dataset.

barycenter_size [int or None (default: None)] Size of the barycenter to generate. If None, the size of the barycenter is that of the data provided at fit time or that of the initial barycenter if specified.

init_barycenter [array or None (default: None)] Initial barycenter to start from for the optimization process.

max_iter [int (default: 30)] Number of iterations of the Expectation-Maximization optimization procedure.

tol [float (default: 1e-5)] Tolerance to use for early stopping: if the decrease in cost is lower than this value, the Expectation-Maximization procedure stops.

weights: None or array Weights of each X[i]. Must be the same size as len(X). If None, uniform weights are used.

metric_params: dict or None (default: None) DTW constraint parameters to be used. See *tslearn.metrics.dtw_path* for a list of accepted parameters If None, no constraint is used for DTW computations.

verbose [boolean (default: False)] Whether to print information about the cost at each iteration or not.

n_init [int (default: 1)] Number of different initializations to be tried (useful only is init_barycenter is set to None, otherwise, all trials will reach the same performance)

Returns

numpy.array of shape (barycenter_size, d) or (sz, d) if barycenter_size is None DBA barycenter of the provided time series dataset.

References

[1], [2]

Examples

```
>>> time_series = [[1, 2, 3, 4], [1, 2, 4, 5]]
>>> dtw_barycenter_averaging(time_series, max_iter=5)
array([[1. ],
       [2. ],
       [3.5],
       [4.5]])
>>> time_series = [[1, 2, 3, 4], [1, 2, 3, 4, 5]]
>>> dtw_barycenter_averaging(time_series, max_iter=5)
array([[1. ],
       [2. ],
       [3. ],
       [4. ],
       [4.5]])
>>> dtw_barycenter_averaging(time_series, max_iter=5,
...                          metric_params={"itakura_max_slope": 2})
array([[1. ],
       [2. ],
       [3. ],
       [3.5],
       [4.5]])
>>> dtw_barycenter_averaging(time_series, max_iter=5, barycenter_size=3)
array([[1.5       ],
       [3.        ],
       [4.33333333]])
>>> dtw_barycenter_averaging([[0, 0, 0], [10, 10, 10]], max_iter=1,
...                          weights=numpy.array([0.75, 0.25]))
array([[2.5],
       [2.5],
       [2.5]])
```

Examples using `tslearn.barycenters.dtw_barycenter_averaging`

- *Barycenters*

3.1.3 tslearn.barycenters.dtw_barycenter_averaging_subgradient

tslearn.barycenters.**dtw_barycenter_averaging_subgradient**(*X*, *barycenter_size=None*, *init_barycenter=None*, *max_iter=30*, *initial_step_size=0.05*, *final_step_size=0.005*, *tol=1e-05*, *random_state=None*, *weights=None*, *metric_params=None*, *verbose=False*)

DTW Barycenter Averaging (DBA) method estimated through subgradient descent algorithm.

DBA was originally presented in [1]. This implementation is based on a idea from [2] (Stochastic Subgradient Mean Algorithm).

Parameters

X [array-like, shape=(n_ts, sz, d)] Time series dataset.

barycenter_size [int or None (default: None)] Size of the barycenter to generate. If None, the size of the barycenter is that of the data provided at fit time or that of the initial barycenter if specified.

init_barycenter [array or None (default: None)] Initial barycenter to start from for the optimization process.

max_iter [int (default: 30)] Number of iterations of the Expectation-Maximization optimization procedure.

initial_step_size [float (default: 0.05)] Initial step size for the subgradient descent algorithm. Default value is the one suggested in [2].

final_step_size [float (default: 0.005)] Final step size for the subgradient descent algorithm. Default value is the one suggested in [2].

tol [float (default: 1e-5)] Tolerance to use for early stopping: if the decrease in cost is lower than this value, the Expectation-Maximization procedure stops.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

weights: None or array Weights of each X[i]. Must be the same size as len(X). If None, uniform weights are used.

metric_params: dict or None (default: None) DTW constraint parameters to be used. See *tslearn.metrics.dtw_path* for a list of accepted parameters If None, no constraint is used for DTW computations.

verbose [boolean (default: False)] Whether to print information about the cost at each iteration or not.

Returns

numpy.array of shape (barycenter_size, d) or (sz, d) if barycenter_size is None DBA barycenter of the provided time series dataset.

References

[1], [2]

Examples

```
>>> time_series = [[1, 2, 3, 4], [1, 2, 4, 5]]
>>> dtw_barycenter_averaging_subgradient(
...     time_series,
...     max_iter=10,
...     random_state=0
... )  # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
array([[1. ],
       [2. ],
       [3.5...],
       [4.5...]])
```

Examples using `tslearn.barycenters.dtw_barycenter_averaging_subgradient`

- *Barycenters*

3.1.4 tslearn.barycenters.softdtw_barycenter

`tslearn.barycenters.`**`softdtw_barycenter`**(*X*, *gamma=1.0*, *weights=None*, *method='L-BFGS-B'*, *tol=0.001*, *max_iter=50*, *init=None*)

Compute barycenter (time series averaging) under the soft-DTW [1] geometry.

Soft-DTW was originally presented in [1].

Parameters

X [array-like, shape=(n_ts, sz, d)] Time series dataset.

gamma: float Regularization parameter. Lower is less smoothed (closer to true DTW).

weights: None or array Weights of each X[i]. Must be the same size as len(X). If None, uniform weights are used.

method: string Optimization method, passed to *scipy.optimize.minimize*. Default: L-BFGS.

tol: float Tolerance of the method used.

max_iter: int Maximum number of iterations.

init: array or None (default: None) Initial barycenter to start from for the optimization process. If *None*, euclidean barycenter is used as a starting point.

Returns

numpy.array of shape (bsz, d) where *bsz* is the size of the *init* array if provided or *sz* otherwise Soft-DTW barycenter of the provided time series dataset.

References

[1]

Examples

```
>>> time_series = [[1, 2, 3, 4], [1, 2, 4, 5]]
>>> softdtw_barycenter(time_series, max_iter=5)
array([[1.25161574],
       [2.03821705],
       [3.5101956 ],
       [4.36140605]])
>>> time_series = [[1, 2, 3, 4], [1, 2, 3, 4, 5]]
>>> softdtw_barycenter(time_series, max_iter=5)
array([[1.21349933],
       [1.8932251 ],
       [2.67573269],
       [3.51057026],
       [4.33645802]])
```

Examples using `tslearn.barycenters.softdtw_barycenter`

- *Barycenters*
- *Soft-DTW weighted barycenters*

3.2 tslearn.clustering

The *tslearn.clustering* module gathers time series specific clustering algorithms.

User guide: See the *Clustering* section for further details.

Classes

KernelKMeans([n_clusters, kernel, max_iter, ...])	Kernel K-means.
KShape([n_clusters, max_iter, tol, n_init, ...])	KShape clustering for time series.
TimeSeriesKMeans([n_clusters, max_iter, ...])	K-means clustering for time-series data.

3.2.1 tslearn.clustering.KernelKMeans

class tslearn.clustering.**KernelKMeans**(*n_clusters=3*, *kernel='gak'*, *max_iter=50*, *tol=1e-06*, *n_init=1*, *kernel_params=None*, *sigma=1.0*, *n_jobs=None*, *verbose=0*, *random_state=None*)

Kernel K-means.

Parameters

n_clusters [int (default: 3)] Number of clusters to form.

kernel [string, or callable (default: "gak")] The kernel should either be "gak", in which case the Global Alignment Kernel from [2] is used or a value that is accepted as a metric by scikit-learn's pairwise_kernels

max_iter [int (default: 50)] Maximum number of iterations of the k-means algorithm for a single run.

tol [float (default: 1e-6)] Inertia variation threshold. If at some point, inertia varies less than this

threshold between two consecutive iterations, the model is considered to have converged and the algorithm stops.

n_init [int (default: 1)] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

kernel_params [dict or None (default: None)] Kernel parameters to be passed to the kernel function. None means no kernel parameter is set. For Global Alignment Kernel, the only parameter of interest is *sigma*. If set to 'auto', it is computed based on a sampling of the training set (cf *tslearn.metrics.sigma_gak*). If no specific value is set for *sigma*, its defaults to 1.

sigma [float or "auto" (default: "auto")] Bandwidth parameter for the Global Alignment kernel. If set to 'auto', it is computed based on a sampling of the training set (cf *tslearn.metrics.sigma_gak*)

Deprecated since version 0.4: Setting *sigma* directly as a parameter for KernelKMeans and GlobalAlignmentKernelKMeans is deprecated in version 0.4 and will be removed in 0.6. Use *kernel_params* instead.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for GAK cross-similarity matrix computations. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose [int (default: 0)] If nonzero, joblib progress messages are printed.

random_state [integer or numpy.RandomState, optional] Generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Attributes

labels_ [numpy.ndarray] Labels of each point

inertia_ [float] Sum of distances of samples to their closest cluster center (computed using the kernel trick).

sample_weight_ [numpy.ndarray] The weight given to each sample from the data provided to fit.

n_iter_ [int] The number of iterations performed during fit.

Notes

The training data are saved to disk if this model is serialized and may result in a large model file if the training dataset is large.

References

[1], [2]

Examples

```
>>> from tslearn.generators import random_walks
>>> X = random_walks(n_ts=50, sz=32, d=1)
>>> gak_km = KernelKMeans(n_clusters=3, kernel="gak", random_state=0)
>>> gak_km.fit(X)  # doctest: +ELLIPSIS
KernelKMeans(...)
>>> print(numpy.unique(gak_km.labels_))
[0 1 2]
```

Methods

fit(X[, y, sample_weight])	Compute kernel k-means clustering.
fit_predict(X[, y])	Fit kernel k-means clustering using X and then predict the closest cluster each time series in X belongs to.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each time series in X belongs to.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.

fit (*X*, *y=None*, *sample_weight=None*)
Compute kernel k-means clustering.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

y Ignored

sample_weight [array-like of shape=(n_ts,) or None (default: None)] Weights to be given to time series in the learning process. By default, all time series weights are equal.

fit_predict (*X*, *y=None*)
Fit kernel k-means clustering using X and then predict the closest cluster each time series in X belongs to.

It is more efficient to use this method than to sequentially call fit and predict.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset to predict.

y Ignored

Returns

labels [array of shape=(n_ts,)] Index of the cluster each sample belongs to.

classmethod from_hdf5 (*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*X*)
Predict the closest cluster each time series in X belongs to.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset to predict.

Returns

labels [array of shape=(n_ts,)] Index of the cluster each sample belongs to.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

Examples using `tslearn.clustering.KernelKMeans`

- *Kernel k-means*

3.2.2 tslearn.clustering.KShape

class tslearn.clustering.**KShape**(*n_clusters=3*, *max_iter=100*, *tol=1e-06*, *n_init=1*, *verbose=False*, *random_state=None*, *init='random'*)
KShape clustering for time series.

KShape was originally presented in [1].

Parameters

n_clusters [int (default: 3)] Number of clusters to form.

max_iter [int (default: 100)] Maximum number of iterations of the k-Shape algorithm.

tol [float (default: 1e-6)] Inertia variation threshold. If at some point, inertia varies less than this threshold between two consecutive iterations, the model is considered to have converged and the algorithm stops.

n_init [int (default: 1)] Number of time the k-Shape algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

verbose [bool (default: False)] Whether or not to print information about the inertia while learning the model.

random_state [integer or numpy.RandomState, optional] Generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

init [{'random' or ndarray} (default: 'random')] Method for initialization. 'random': choose k observations (rows) at random from data for the initial centroids. If an ndarray is passed, it should be of shape (n_clusters, ts_size, d) and gives the initial centers.

Attributes

cluster_centers_ [numpy.ndarray of shape (sz, d).] Centroids

labels_ [numpy.ndarray of integers with shape (n_ts,).] Labels of each point

inertia_ [float] Sum of distances of samples to their closest cluster center.

n_iter_ [int] The number of iterations performed during fit.

Notes

This method requires a dataset of equal-sized time series.

References

[1]

Examples

```
>>> from tslearn.generators import random_walks
>>> X = random_walks(n_ts=50, sz=32, d=1)
>>> X = TimeSeriesScalerMeanVariance(mu=0., std=1.).fit_transform(X)
>>> ks = KShape(n_clusters=3, n_init=1, random_state=0).fit(X)
>>> ks.cluster_centers_.shape
(3, 32, 1)
```

Methods

fit(X[, y])	Compute k-Shape clustering.
fit_predict(X[, y])	Fit k-Shape clustering using X and then predict the closest cluster each time series in X belongs to.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each time series in X belongs to.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.

fit (*X*, *y=None*)
Compute k-Shape clustering.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

y Ignored

fit_predict (*X*, *y=None*)
Fit k-Shape clustering using X and then predict the closest cluster each time series in X belongs to.

It is more efficient to use this method than to sequentially call fit and predict.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset to predict.

y Ignored

Returns

labels [array of shape=(n_ts,)] Index of the cluster each sample belongs to.

classmethod from_hdf5(*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*X*)
Predict the closest cluster each time series in X belongs to.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset to predict.

Returns

labels [array of shape=(n_ts,)] Index of the cluster each sample belongs to.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

Examples using `tslearn.clustering.KShape`

- *KShape*
- *Model Persistence*

3.2.3 `tslearn.clustering`.TimeSeriesKMeans

class tslearn.clustering.**TimeSeriesKMeans**(*n_clusters=3*, *max_iter=50*, *tol=1e-06*, *n_init=1*, *metric='euclidean'*, *max_iter_barycenter=100*, *metric_params=None*, *n_jobs=None*, *dtw_inertia=False*, *verbose=0*, *random_state=None*, *init='k-means++'*)

K-means clustering for time-series data.

Parameters

n_clusters [int (default: 3)] Number of clusters to form.

max_iter [int (default: 50)] Maximum number of iterations of the k-means algorithm for a single run.

tol [float (default: 1e-6)] Inertia variation threshold. If at some point, inertia varies less than this threshold between two consecutive iterations, the model is considered to have converged and the algorithm stops.

n_init [int (default: 1)] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

metric [{"euclidean", "dtw", "softdtw"} (default: "euclidean")] Metric to be used for both cluster assignment and barycenter computation. If "dtw", DBA is used for barycenter computation.

max_iter_barycenter [int (default: 100)] Number of iterations for the barycenter computation process. Only used if *metric="dtw"* or *metric="softdtw"*.

metric_params [dict or None (default: None)] Parameter values for the chosen metric. For metrics that accept parallelization of the cross-distance matrix computations, *n_jobs* key passed in *metric_params* is overridden by the *n_jobs* argument.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for cross-distance matrix computations. Ignored if the cross-distance matrix cannot be computed using parallelization. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

dtw_inertia: bool (default: False) Whether to compute DTW inertia even if DTW is not the chosen metric.

verbose [int (default: 0)] If nonzero, print information about the inertia while learning the model and joblib progress messages are printed.

random_state [integer or numpy.RandomState, optional] Generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

init [{'k-means++', 'random' or an ndarray} (default: 'k-means++')] Method for initialization: 'k-means++' : use k-means++ heuristic. See scikit-learn's k_init_ for more. 'random': choose k observations (rows) at random from data for the initial centroids. If an ndarray is passed, it should be of shape (n_clusters, ts_size, d) and gives the initial centers.

Attributes

labels_ [numpy.ndarray] Labels of each point.

cluster_centers_ [numpy.ndarray of shape (n_clusters, sz, d)] Cluster centers. *sz* is the size of the time series used at fit time if the init method is 'k-means++' or 'random', and the size of the longest initial centroid if those are provided as a numpy array through init parameter.

inertia_ [float] Sum of distances of samples to their closest cluster center.

n_iter_ [int] The number of iterations performed during fit.

Notes

If *metric* is set to *"euclidean"*, the algorithm expects a dataset of equal-sized time series.

Examples

```
>>> from tslearn.generators import random_walks
>>> X = random_walks(n_ts=50, sz=32, d=1)
>>> km = TimeSeriesKMeans(n_clusters=3, metric="euclidean", max_iter=5,
...                       random_state=0).fit(X)
>>> km.cluster_centers_.shape
```

(continues on next page)

(continued from previous page)

```
(3, 32, 1)
>>> km_dba = TimeSeriesKMeans(n_clusters=3, metric="dtw", max_iter=5,
...                           max_iter_barycenter=5,
...                           random_state=0).fit(X)
>>> km_dba.cluster_centers_.shape
(3, 32, 1)
>>> km_sdtw = TimeSeriesKMeans(n_clusters=3, metric="softdtw", max_iter=5,
...                            max_iter_barycenter=5,
...                            metric_params={"gamma": .5},
...                            random_state=0).fit(X)
>>> km_sdtw.cluster_centers_.shape
(3, 32, 1)
>>> X_bis = to_time_series_dataset([[1, 2, 3, 4],
...                                 [1, 2, 3],
...                                 [2, 5, 6, 7, 8, 9]])
>>> km = TimeSeriesKMeans(n_clusters=2, max_iter=5,
...                       metric="dtw", random_state=0).fit(X_bis)
>>> km.cluster_centers_.shape
(2, 6, 1)
```

Methods

fit(X[, y])	Compute k-means clustering.
fit_predict(X[, y])	Fit k-means clustering using X and then predict the closest cluster each time series in X belongs to.
fit_transform(X[, y])	Fit to data, then transform it.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each time series in X belongs to.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X)	Transform X to a cluster-distance space.

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

Compute k-means clustering.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

y Ignored

fit_predict (*X*, *y=None*)

Fit k-means clustering using X and then predict the closest cluster each time series in X belongs to.

It is more efficient to use this method than to sequentially call fit and predict.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset to predict.

y Ignored

Returns

labels [array of shape=(n_ts,)] Index of the cluster each sample belongs to.

fit_transform(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [{array-like, sparse matrix, dataframe} of shape (n_samples, n_features)]

y [ndarray of shape (n_samples,), default=None] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [ndarray array of shape (n_samples, n_features_new)] Transformed array.

classmethod from_hdf5(*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*X*)
Predict the closest cluster each time series in X belongs to.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset to predict.

Returns

labels [array of shape=(n_ts,)] Index of the cluster each sample belongs to.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

transform(*X*)
Transform X to a cluster-distance space.

In the new space, each dimension is the distance to the cluster centers.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset

Returns

distances [array of shape=(n_ts, n_clusters)] Distances to cluster centers

Examples using `tslearn.clustering.TimeSeriesKMeans`

- *k-means*

Functions

silhouette_score(X, labels[, metric, ...])	Compute the mean Silhouette Coefficient of all samples (cf.

3.2.4 tslearn.clustering.silhouette_score

tslearn.clustering.**silhouette_score**(*X*, *labels*, *metric=None*, *sample_size=None*, *metric_params=None*, *n_jobs=None*, *verbose=0*, *random_state=None*, ***kwds*)

Compute the mean Silhouette Coefficient of all samples (cf. [1] and [2]).

Read more in the scikit-learn documentation.

Parameters

X [array [n_ts, n_ts] if metric == "precomputed", or, [n_ts, sz, d] otherwise] Array of pairwise distances between time series, or a time series dataset.

labels [array, shape = [n_ts]] Predicted labels for each time series.

metric [string, callable or None (default: None)] The metric to use when calculating distance between time series. Should be one of {'dtw', 'softdtw', 'euclidean'} or a callable distance function or None. If 'softdtw' is passed, a normalized version of Soft-DTW is used that is defined as *sdtw_(x,y) := sdtw(x,y) - 1/2(sdtw(x,x)+sdtw(y,y))*. If X is the distance array itself, use `metric="precomputed"`. If None, dtw is used.

sample_size [int or None (default: None)] The size of the sample to use when computing the Silhouette Coefficient on a random subset of the data. If `sample_size is None`, no sampling is used.

metric_params [dict or None (default: None)] Parameter values for the chosen metric. For metrics that accept parallelization of the cross-distance matrix computations, *n_jobs* key passed in *metric_params* is overridden by the *n_jobs* argument.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for cross-distance matrix computations. Ignored if the cross-distance matrix cannot be computed using parallelization. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose [int (default: 0)] If nonzero, print information about the inertia while learning the model and joblib progress messages are printed.

random_state [int, RandomState instance or None, optional (default: None)] The generator used to randomly select a subset of samples. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used when `sample_size is not None`.

****kwds** [optional keyword parameters] Any further parameters are passed directly to the distance function, just as for the *metric_params* parameter.

Returns

silhouette [float] Mean Silhouette Coefficient for all samples.

References

[1], [2]

Examples

```
>>> from tslearn.generators import random_walks
>>> from tslearn.metrics import cdist_dtw
>>> numpy.random.seed(0)
>>> X = random_walks(n_ts=20, sz=16, d=1)
>>> labels = numpy.random.randint(2, size=20)
>>> silhouette_score(X, labels, metric="dtw")  # doctest: +ELLIPSIS
0.13383800...
>>> silhouette_score(X, labels, metric="euclidean")  # doctest: +ELLIPSIS
0.09126917...
>>> silhouette_score(X, labels, metric="softdtw")  # doctest: +ELLIPSIS
0.17953934...
>>> silhouette_score(X, labels, metric="softdtw",
...                  metric_params={"gamma": 2.})     # doctest: +ELLIPSIS
0.17591060...
>>> silhouette_score(cdist_dtw(X), labels,
...                  metric="precomputed")  # doctest: +ELLIPSIS
0.13383800...
```

3.3 tslearn.datasets

The *tslearn.datasets* module provides simplified access to standard time series datasets.

Classes

UCR_UEA_datasets([use_cache])	A convenience class to access UCR/UEA time series datasets.
CachedDatasets()	A convenience class to access cached time series datasets.

3.3.1 tslearn.datasets.UCR_UEA_datasets

class tslearn.datasets.**UCR_UEA_datasets**(*use_cache=True*)

A convenience class to access UCR/UEA time series datasets.

When using one (or several) of these datasets in research projects, please cite [1].

Parameters

use_cache [bool (default: True)] Whether a cached version of the dataset should be used, if found.

Notes

Downloading dataset files can be time-consuming, it is recommended using *use_cache=True* (default) in order to only experience downloading time once per dataset and work on a cached version of the datasets after it.

References

[1]

Methods

baseline_accuracy([list_datasets, list_methods])	Report baseline performances as provided by UEA/UCR website (for univariate datasets only).
cache_all()	Cache all datasets from the UCR/UEA archive for later use.
list_cached_datasets()	List datasets from the UCR/UEA archive that are available in cache.
list_datasets()	List datasets (both univariate and multivariate) available in the UCR/UEA archive.
list_multivariate_datasets()	List multivariate datasets in the UCR/UEA archive.
list_univariate_datasets()	List univariate datasets in the UCR/UEA archive.
load_dataset(dataset_name)	Load a dataset from the UCR/UEA archive from its name.

baseline_accuracy(*list_datasets=None*, *list_methods=None*)

Report baseline performances as provided by UEA/UCR website (for univariate datasets only).

Parameters

list_datasets: list or None (default: None) A list of strings indicating for which datasets performance should be reported. If None, performance is reported for all datasets.

list_methods: list or None (default: None) A list of baselines methods for which performance should be reported. If None, performance for all baseline methods is reported.

Returns

dict A dictionary in which keys are dataset names and associated values are themselves dictionaries that provide accuracy scores for the requested methods.

Examples

```
>>> uea_ucr = UCR_UEA_datasets()
>>> dict_acc = uea_ucr.baseline_accuracy(
...         list_datasets=["Adiac", "ChlorineConcentration"],
...         list_methods=["C45"])
>>> len(dict_acc)
2
>>> dict_acc["Adiac"]  # doctest: +ELLIPSIS
{'C45': 0.542199...}
>>> dict_acc = uea_ucr.baseline_accuracy()
>>> len(dict_acc)
85
```

cache_all()

Cache all datasets from the UCR/UEA archive for later use.

list_cached_datasets()

List datasets from the UCR/UEA archive that are available in cache.

Examples

```
>>> beetlefly = UCR_UEA_datasets().load_dataset("BeetleFly")
>>> l = UCR_UEA_datasets().list_cached_datasets()
>>> "BeetleFly" in l
True
```

list_datasets()
List datasets (both univariate and multivariate) available in the UCR/UEA archive.

Examples

```
>>> l = UCR_UEA_datasets().list_datasets()
>>> "PenDigits" in l
True
>>> "BeetleFly" in l
True
>>> "DatasetThatDoesNotExist" in l
False
```

list_multivariate_datasets()
List multivariate datasets in the UCR/UEA archive.

Examples

```
>>> l = UCR_UEA_datasets().list_multivariate_datasets()
>>> "PenDigits" in l
True
```

list_univariate_datasets()
List univariate datasets in the UCR/UEA archive.

Examples

```
>>> l = UCR_UEA_datasets().list_univariate_datasets()
>>> len(l)
85
```

load_dataset(*dataset_name*)
Load a dataset from the UCR/UEA archive from its name.

Parameters

dataset_name [str] Name of the dataset. Should be in the list returned by *list_datasets*

Returns

numpy.ndarray of shape (n_ts_train, sz, d) or None Training time series. None if unsuccessful.

numpy.ndarray of integers or strings with shape (n_ts_train,) or None Training labels. None if unsuccessful.

numpy.ndarray of shape (n_ts_test, sz, d) or None Test time series. None if unsuccessful.

numpy.ndarray of integers or strings with shape (n_ts_test,) or None Test labels. None if unsuccessful.

Examples

```
>>> data_loader = UCR_UEA_datasets()
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
...         "TwoPatterns")
>>> X_train.shape
(1000, 128, 1)
>>> y_train.shape
(1000,)
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
...         "StarLightCurves")
>>> X_train.shape
(1000, 1024, 1)
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
...         "CinCECGTorso")
>>> X_train.shape
(40, 1639, 1)
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
...         "PenDigits")
>>> X_train.shape
(7494, 8, 2)
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
...         "StarlightCurves")
>>> X_train.shape
(1000, 1024, 1)
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
...         "DatasetThatDoesNotExist")
>>> print(X_train)
None
```

Examples using `tslearn.datasets.UCR_UEA_datasets`

- *1-NN with SAX + MINDIST*
- *Early Classification*

3.3.2 `tslearn.datasets`.CachedDatasets

class tslearn.datasets.**CachedDatasets**

A convenience class to access cached time series datasets.

When using the Trace dataset, please cite [1].

References

[1]

Methods

list_datasets()	List cached datasets.
load_dataset(dataset_name)	Load a cached dataset from its name.

list_datasets()
List cached datasets.

load_dataset(*dataset_name*)
Load a cached dataset from its name.

Parameters

dataset_name [str] Name of the dataset. Should be in the list returned by *list_datasets*

Returns

numpy.ndarray of shape (n_ts_train, sz, d) or None Training time series. None if unsuccessful.

numpy.ndarray of integers with shape (n_ts_train,) or None Training labels. None if unsuccessful.

numpy.ndarray of shape (n_ts_test, sz, d) or None Test time series. None if unsuccessful.

numpy.ndarray of integers with shape (n_ts_test,) or None Test labels. None if unsuccessful.

Examples

```
>>> data_loader = CachedDatasets()
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
...                                         "Trace")
>>> print(X_train.shape)
(100, 275, 1)
>>> print(y_train.shape)
(100,)
```

Examples using `tslearn.datasets.CachedDatasets`

- *k-NN search*
- *Hyper-parameter tuning of a Pipeline with KNeighborsTimeSeriesClassifier*
- *KShape*
- *Kernel k-means*
- *Barycenters*
- *Soft-DTW weighted barycenters*
- *k-means*
- *SVM and GAK*
- *Learning Shapelets*
- *Aligning discovered shapelets with timeseries*
- *Learning Shapelets: decision boundaries in 2D distance space*
- *Model Persistence*
- *Distance and Matrix Profiles*

3.4 tslearn.early_classification

The *tslearn.early_classification* module gathers early classifiers for time series.

Such classifiers aim at performing prediction as early as possible (i.e. they do not necessarily wait for the end of the series before prediction is triggered).

User guide: See the *Early Classification* section for further details.

Classes

NonMyopicEarlyClassifier([n_clusters, ...])	Early Classification modelling for time series using the model presented in [1].

3.4.1 tslearn.early_classification.NonMyopicEarlyClassifier

class tslearn.early_classification.**NonMyopicEarlyClassifier**(*n_clusters=2*, *base_classifier=None*, *min_t=1*, *lamb=1.0*, *cost_time_parameter=1.0*, *random_state=None*)

Early Classification modelling for time series using the model presented in [1].

Parameters

n_clusters [int] Number of clusters to form.

base_classifier [Estimator or None] Estimator (instance) to be cloned and used for classifications. If None, the chosen classifier is a 1NN with Euclidean metric.

min_t [int] Earliest time at which a classification can be performed on a time series

lamb [float] Value of the hyper parameter lambda used during the computation of the cost function to evaluate the probability that a time series belongs to a cluster given the time series.

cost_time_parameter [float] Parameter of the cost function of time. This function is of the form : f(time) = time * cost_time_parameter

random_state: int Random state of the base estimator

Attributes

classifiers_ [list] A list containing all the classifiers trained for the model, that is, (maximum_time_stamp - min_t) elements.

pyhatyck_ [array like of shape (maximum_time_stamp - min_t, n_cluster, __n_classes, __n_classes)] Contains the probabilities of being classified as class y_hat given class y and cluster ck for a trained classifier. The penultimate dimension of the array is associated to the true class of the series and the last dimension to the predicted class.

pyck_ [array like of shape (__n_classes, n_cluster)] Contains the probabilities of being of true class y given a cluster ck

X_fit_dims [tuple of the same shape as the training dataset]

References

[1]

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [3, 2, 1, 1, 2, 3],
...                                   [3, 2, 1, 1, 2, 3]])
>>> y = [0, 0, 0, 1, 1, 1, 0, 0]
>>> model = NonMyopicEarlyClassifier(n_clusters=3, lamb=1000.,
...                                  cost_time_parameter=.1,
...                                  random_state=0)
>>> model.fit(dataset, y)  # doctest: +ELLIPSIS
NonMyopicEarlyClassifier(...)
>>> print(type(model.classifiers_))
<class 'dict'>
>>> print(model.pyck_)
[[0. 1. 1.]
 [1. 0. 0.]]
>>> preds, pred_times = model.predict_class_and_earliness(dataset)
>>> preds
array([0, 0, 0, 1, 1, 1, 0, 0])
>>> pred_times
array([4, 4, 4, 4, 4, 4, 1, 1])
>>> pred_probas, pred_times = model.predict_proba_and_earliness(dataset)
>>> pred_probas
array([[1., 0.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [1., 0.],
       [1., 0.]])
>>> pred_times
array([4, 4, 4, 4, 4, 4, 1, 1])
```

Methods

early_classification_cost(X, y)	Compute early classification score.
fit(X, y)	Fit early classifier.
get_cluster_probas(Xi)	Compute cluster probability $P(c_k \| Xi)$.
get_params([deep])	Get parameters for this estimator.
predict(X)	Provide predicted class.
predict_class_and_earliness(X)	Provide predicted class as well as prediction timestamps.
predict_proba(X)	Probability estimates.

Continued on next page

Table 12 – continued from previous page

predict_proba_and_earliness(X)	Provide probability estimates as well as prediction timestamps.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

early_classification_cost (*X*, *y*)
Compute early classification score.

The score is computed as:

$$1 - acc + \alpha \frac{1}{n} \sum_i t_i$$

where α is the trade-off parameter (*self.cost_time_parameter*) and t_i are prediction timestamps.

Parameters

X [array-like of shape (n_series, n_timestamps, n_features)] Vector to be scored, where *n_series* is the number of time series, *n_timestamps* is the number of timestamps in the series and *n_features* is the number of features recorded at each timestamp.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True labels for X.

Returns

float Early classification cost (a positive number, the lower the better)

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [3, 2, 1, 1, 2, 3],
...                                   [3, 2, 1, 1, 2, 3]])
>>> y = [0, 0, 0, 1, 1, 1, 0, 0]
>>> model = NonMyopicEarlyClassifier(n_clusters=3, lamb=1000.,
...                                  cost_time_parameter=.1,
...                                  random_state=0)
>>> model.fit(dataset, y)  # doctest: +ELLIPSIS
NonMyopicEarlyClassifier(...)
>>> preds, pred_times = model.predict_class_and_earliness(dataset)
>>> preds
array([0, 0, 0, 1, 1, 1, 0, 0])
>>> pred_times
array([4, 4, 4, 4, 4, 4, 1, 1])
>>> model.early_classification_cost(dataset, y)
0.325
```

fit (*X*, *y*)
Fit early classifier.

Parameters

X [array-like of shape (n_series, n_timestamps, n_features)] Training data, where *n_series* is the number of time series, *n_timestamps* is the number of timestamps in the series and *n_features* is the number of features recorded at each timestamp.

y [array-like of shape (n_samples,)] Target values. Will be cast to X's dtype if necessary

Returns

self [returns an instance of self.]

get_cluster_probas(*Xi*)

Compute cluster probability $P(c_k|Xi)$.

This quantity is computed using the following formula:

$$P(c_k|Xi) = \frac{s_k(Xi)}{\sum_j s_j(Xi)}$$

where

$$s_k(Xi) = \frac{1}{1 + \exp -\lambda\Delta_k(Xi)}$$

with

$$\Delta_k(Xi) = \frac{\bar{D} - d(Xi, c_k)}{\bar{D}}$$

and $\bar{D}$ is the average of the distances between *Xi* and the cluster centers.

Parameters

Xi: numpy array, shape (t, d) A time series observed up to time t

Returns

probas [numpy array, shape (n_clusters,)]

Examples

```
>>> from tslearn.utils import to_time_series
>>> dataset = to_time_series_dataset([[1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [3, 2, 1, 1, 2, 3],
...                                   [3, 2, 1, 1, 2, 3]])
>>> y = [0, 0, 0, 1, 1, 1, 0, 0]
>>> ts0 = to_time_series([1, 2])
>>> model = NonMyopicEarlyClassifier(n_clusters=3, lamb=0.,
...                                  random_state=0)
>>> probas = model.fit(dataset, y).get_cluster_probas(ts0)
>>> probas.shape
(3,)
>>> probas  # doctest: +ELLIPSIS
array([0.33..., 0.33..., 0.33...])
>>> model = NonMyopicEarlyClassifier(n_clusters=3, lamb=10000.,
...                                  random_state=0)
```

(continues on next page)

(continued from previous page)

```
>>> probas = model.fit(dataset, y).get_cluster_probas(ts0)
>>> probas.shape
(3,)
>>> probas
array([0.5, 0.5, 0. ])
>>> ts1 = to_time_series([3, 2])
>>> model.get_cluster_probas(ts1)
array([0., 0., 1.])
```

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*X*)
Provide predicted class.

Parameters

X [array-like of shape (n_series, n_timestamps, n_features)] Vector to be scored, where *n_series* is the number of time series, *n_timestamps* is the number of timestamps in the series and *n_features* is the number of features recorded at each timestamp.

Returns

array, shape (n_samples,) Predicted classes.

predict_class_and_earliness(*X*)
Provide predicted class as well as prediction timestamps.

Prediction timestamps are timestamps at which a prediction is made in early classification setting.

Parameters

X [array-like of shape (n_series, n_timestamps, n_features)] Vector to be scored, where *n_series* is the number of time series, *n_timestamps* is the number of timestamps in the series and *n_features* is the number of features recorded at each timestamp.

Returns

array, shape (n_samples,) Predicted classes.

array-like of shape (n_series,) Prediction timestamps.

predict_proba(*X*)
Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

Parameters

X [array-like of shape (n_series, n_timestamps, n_features)] Vector to be scored, where *n_series* is the number of time series, *n_timestamps* is the number of timestamps in the series and *n_features* is the number of features recorded at each timestamp.

Returns

array-like of shape (n_series, n_classes) Probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

predict_proba_and_earliness(*X*)
Provide probability estimates as well as prediction timestamps.

Prediction timestamps are timestamps at which a prediction is made in early classification setting. The returned estimates for all classes are ordered by the label of classes.

Parameters

X [array-like of shape (n_series, n_timestamps, n_features)] Vector to be scored, where *n_series* is the number of time series, *n_timestamps* is the number of timestamps in the series and *n_features* is the number of features recorded at each timestamp.

Returns

array-like of shape (n_series, n_classes) Probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

array-like of shape (n_series,) Prediction timestamps.

score(*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

Examples using `tslearn.early_classification.NonMyopicEarlyClassifier`

- *Early Classification*

3.5 tslearn.generators

The *tslearn.generators* module gathers synthetic time series dataset generation routines.

Functions

random_walk_blobs([n_ts_per_blob, sz, d, ...])	Blob-based random walk time series generator.
random_walks([n_ts, sz, d, mu, std, ...])	Random walk time series generator.

3.5.1 tslearn.generators.random_walk_blobs

tslearn.generators.**random_walk_blobs**(*n_ts_per_blob=100*, *sz=256*, *d=1*, *n_blobs=2*, *noise_level=1.0*, *random_state=None*)

Blob-based random walk time series generator.

Generate n_ts_per_blobs * n_blobs time series of size sz and dimensionality d. Generated time series follow the model:

$$ts[t] = ts[t-1] + a$$

where a is drawn from a normal distribution of mean mu and standard deviation std.

Each blob contains time series derived from a same seed time series with added white noise.

Parameters

n_ts_per_blob [int (default: 100)] Number of time series in each blob

sz [int (default: 256)] Length of time series (number of time instants)

d [int (default: 1)] Dimensionality of time series

n_blobs [int (default: 2)] Number of blobs

noise_level [float (default: 1.)] Standard deviation of white noise added to time series in each blob

random_state [integer or numpy.RandomState or None (default: None)] Generator used to draw the time series. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Returns

numpy.ndarray A dataset of random walk time series

numpy.ndarray Labels associated to random walk time series (blob id)

Examples

```
>>> X, y = random_walk_blobs(n_ts_per_blob=100, sz=256, d=5, n_blobs=3)
>>> X.shape
(300, 256, 5)
>>> y.shape
(300,)
```

Examples using tslearn.generators.random_walk_blobs

- *Nearest neighbors*

3.5.2 tslearn.generators.random_walks

tslearn.generators.**random_walks**(*n_ts=100*, *sz=256*, *d=1*, *mu=0.0*, *std=1.0*, *random_state=None*)

Random walk time series generator.

Generate n_ts time series of size sz and dimensionality d. Generated time series follow the model:

$$ts[t] = ts[t-1] + a$$

where a is drawn from a normal distribution of mean mu and standard deviation std.

Parameters

n_ts [int (default: 100)] Number of time series.

sz [int (default: 256)] Length of time series (number of time instants).

d [int (default: 1)] Dimensionality of time series.

mu [float (default: 0.)] Mean of the normal distribution from which random walk steps are drawn.

std [float (default: 1.)] Standard deviation of the normal distribution from which random walk steps are drawn.

random_state [integer or numpy.RandomState or None (default: None)] Generator used to draw the time series. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Returns

numpy.ndarray A dataset of random walk time series

Examples

```
>>> random_walks(n_ts=100, sz=256, d=5, mu=0., std=1.).shape
(100, 256, 5)
```

Examples using `tslearn.generators.random_walks`

- *LB_Keogh*
- *sDTW multi path matching*
- *DTW computation with a custom distance metric*
- *PAA and SAX features*

3.6 tslearn.matrix_profile

The *tslearn.matrix_profile* module gathers methods for the computation of Matrix Profiles from time series.

User guide: See the *Matrix Profile* section for further details.

Classes

MatrixProfile([subsequence_length, ...])	Matrix Profile transformation.

3.6.1 tslearn.matrix_profile.MatrixProfile

class tslearn.matrix_profile.**MatrixProfile**(*subsequence_length=1*, *implementation='numpy'*, *scale=True*)

Matrix Profile transformation.

Matrix Profile was originally presented in [1].

Parameters

subsequence_length [int (default: 1)] Length of the subseries (also called window size) to be used for subseries distance computations.

implementation [str (default: "numpy")] Matrix profile implementation to use. Defaults to "numpy" to use the pure numpy version. All the available implementations are ["numpy", "stump", "gpu_stump"].

"stump" and "gpu_stump" are both implementations from the stumpy python library, the latter requiring a GPU. Stumpy is a library for efficiently computing the matrix profile which is optimized for speed, performance and memory. See [2] for the documentation. "numpy" is the default pure numpy implementation and does not require stumpy to be installed.

scale: bool (default: True) Whether input data should be scaled for each feature of each time series to have zero mean and unit variance. Default for this parameter is set to *True* to match the standard matrix profile setup.

References

[1], [2]

Examples

```
>>> time_series = [0., 1., 3., 2., 9., 1., 14., 15., 1., 2., 2., 10., 7.]
>>> ds = [time_series]
>>> mp = MatrixProfile(subsequence_length=4, scale=False)
>>> mp.fit_transform(ds)[0, :, 0]  # doctest: +ELLIPSIS
array([ 6.85...,  1.41...,  6.16...,  7.93..., 11.40...,
       13.56..., 18.  ..., 13.96...,  1.41...,  6.16...])
```

Methods

fit(X[, y])	Fit a Matrix Profile representation.
fit_transform(X[, y])	Transform a dataset of time series into its Matrix Profile
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

Continued on next page

Table 15 – continued from previous page

to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X[, y])	Transform a dataset of time series into its Matrix Profile

fit (*X*, *y=None*)
Fit a Matrix Profile representation.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

MatrixProfile self

fit_transform (*X*, *y=None*, ***fit_params*)

Transform a dataset of time series into its Matrix Profile representation.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

numpy.ndarray of shape (n_ts, output_size, 1) Matrix-Profile-Transformed dataset. *ouput_size* is equal to *sz - subsequence_length + 1*

classmethod from_hdf5 (*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json (*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle (*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

transform(*X*, *y=None*)

Transform a dataset of time series into its Matrix Profile representation.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

numpy.ndarray of shape (n_ts, output_size, 1) Matrix-Profile-Transformed dataset. *ouput_size* is equal to *sz - subsequence_length + 1*

Examples using `tslearn.matrix_profile.MatrixProfile`

- *Matrix Profile*
- *Distance and Matrix Profiles*

3.7 tslearn.metrics

The `tslearn.metrics` module delivers time-series specific metrics to be used at the core of machine learning algorithms.

User guide: See the *Dynamic Time Warping (DTW)* section for further details.

Functions

`cdist_dtw`(dataset1[, dataset2, ...])	Compute cross-similarity matrix using Dynamic Time Warping (DTW) similarity measure.
`cdist_gak`(dataset1[, dataset2, sigma, ...])	Compute cross-similarity matrix using Global Alignment kernel (GAK).
`dtw`(s1, s2[, global_constraint, ...])	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series and return it.
`dtw_path`(s1, s2[, global_constraint, ...])	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series and return both the path and the similarity.
`dtw_path_from_metric`(s1[, s2, metric, ...])	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series using a distance metric defined by the user and return both the path and the similarity.
`dtw_limited_warping_length`(s1, s2, max_length)	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series under an upper bound constraint on the resulting path length and return the similarity cost.
`dtw_path_limited_warping_length`(s1, s2, ...)	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series under an upper bound constraint on the resulting path length and return the path as well as the similarity cost.
`subsequence_path`(acc_cost_mat, idx_path_end)	Compute the optimal path through a accumulated cost matrix given the endpoint of the sequence.
`subsequence_cost_matrix`(subseq, longseq)	Compute the accumulated cost matrix score between a subsequence and a reference time series.
`dtw_subsequence_path`(subseq, longseq)	Compute sub-sequence Dynamic Time Warping (DTW) similarity measure between a (possibly multidimensional) query and a long time series and return both the path and the similarity.
`gak`(s1, s2[, sigma])	Compute Global Alignment Kernel (GAK) between (possibly multidimensional) time series and return it.
`soft_dtw`(ts1, ts2[, gamma])	Compute Soft-DTW metric between two time series.

Continued on next page

Table 16 – continued from previous page

cdist_soft_dtw(dataset1[, dataset2, gamma])	Compute cross-similarity matrix using Soft-DTW metric.
cdist_soft_dtw_normalized(dataset1[, ...])	Compute cross-similarity matrix using a normalized version of the Soft-DTW metric.
lb_envelope(ts[, radius])	Compute time-series envelope as required by LB_Keogh.
lb_keogh(ts_query[, ts_candidate, radius, ...])	Compute LB_Keogh.
sigma_gak(dataset[, n_samples, random_state])	Compute sigma value to be used for GAK.
gamma_soft_dtw(dataset[, n_samples, ...])	Compute gamma value to be used for GAK/Soft-DTW.

3.7.1 tslearn.metrics.cdist_dtw

tslearn.metrics.**cdist_dtw**(*dataset1*, *dataset2=None*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *n_jobs=None*, *verbose=0*)

Compute cross-similarity matrix using Dynamic Time Warping (DTW) similarity measure.

DTW is computed as the Euclidean distance between aligned time series, i.e., if π is the alignment path:

$$DTW(X, Y) = \sqrt{\sum_{(i,j)\in\pi} (X_i - Y_j)^2}$$

DTW was originally presented in [1] and is discussed in more details in our *dedicated user-guide page*.

Parameters

dataset1 [array-like] A dataset of time series

dataset2 [array-like (default: None)] Another dataset of time series. If *None*, self-similarity of *dataset1* is returned.

global_constraint [{"itakura", "sakoe_chiba"} or None (default: None)] Global constraint to restrict admissible paths for DTW.

sakoe_chiba_radius [int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

itakura_max_slope [float or None (default: None)] Maximum slope for the Itakura parallelogram constraint. If None and *global_constraint* is set to "itakura", a maximum slope of 2. is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose [int, optional (default=0)] The verbosity level: if non zero, progress messages are printed. Above 50, the output is sent to stdout. The frequency of the messages increases with the verbosity level. If it more than 10, all iterations are reported. Glossary for more details.

Returns

cdist [numpy.ndarray] Cross-similarity matrix

See also:

dtw Get DTW similarity score

References

[1]

Examples

```
>>> cdist_dtw([[1, 2, 2, 3], [1., 2., 3., 4.]])
array([[0., 1.],
       [1., 0.]])
>>> cdist_dtw([[1, 2, 2, 3], [1., 2., 3., 4.]], [[1, 2, 3], [2, 3, 4, 5]])
array([[0.        , 2.44948974],
       [1.        , 1.41421356]])
```

3.7.2 tslearn.metrics.cdist_gak

tslearn.metrics.**cdist_gak**(*dataset1*, *dataset2=None*, *sigma=1.0*, *n_jobs=None*, *verbose=0*)

Compute cross-similarity matrix using Global Alignment kernel (GAK).

GAK was originally presented in [1].

Parameters

dataset1 A dataset of time series

dataset2 Another dataset of time series

sigma [float (default 1.)] Bandwidth of the internal gaussian kernel used for GAK

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose [int, optional (default=0)] The verbosity level: if non zero, progress messages are printed. Above 50, the output is sent to stdout. The frequency of the messages increases with the verbosity level. If it more than 10, all iterations are reported. Glossary for more details.

Returns

numpy.ndarray Cross-similarity matrix

See also:

gak Compute Global Alignment kernel

References

[1]

Examples

```
>>> cdist_gak([[1, 2, 2, 3], [1., 2., 3., 4.]], sigma=2.)
array([[1.        , 0.65629661],
       [0.65629661, 1.        ]])
>>> cdist_gak([[1, 2, 2], [1., 2., 3., 4.]],
...           [[1, 2, 2, 3], [1., 2., 3., 4.], [1, 2, 2, 3]],
...           sigma=2.)
array([[0.71059484, 0.29722877, 0.71059484],
       [0.65629661, 1.        , 0.65629661]])
```

3.7.3 tslearn.metrics.dtw

tslearn.metrics.**dtw**(*s1*, *s2*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*)

Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series and return it.

DTW is computed as the Euclidean distance between aligned time series, i.e., if π is the optimal alignment path:

$$DTW(X, Y) = \sqrt{\sum_{(i,j)\in\pi} \|X_i - Y_j\|^2}$$

Note that this formula is still valid for the multivariate case.

It is not required that both time series share the same size, but they must be the same dimension. DTW was originally presented in [1] and is discussed in more details in our *dedicated user-guide page*.

Parameters

s1 A time series.

s2 Another time series.

global_constraint [{"itakura", "sakoe_chiba"} or None (default: None)] Global constraint to restrict admissible paths for DTW.

sakoe_chiba_radius [int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

itakura_max_slope [float or None (default: None)] Maximum slope for the Itakura parallelogram constraint. If None and *global_constraint* is set to "itakura", a maximum slope of 2. is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

Returns

float Similarity score

See also:

dtw_path Get both the matching path and the similarity score for DTW

cdist_dtw Cross similarity matrix between time series datasets

References

[1]

Examples

```
>>> dtw([1, 2, 3], [1., 2., 2., 3.])
0.0
>>> dtw([1, 2, 3], [1., 2., 2., 3., 4.])
1.0
```

3.7.4 tslearn.metrics.dtw_path

tslearn.metrics.**dtw_path**(*s1*, *s2*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*)

Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series and return both the path and the similarity.

DTW is computed as the Euclidean distance between aligned time series, i.e., if π is the alignment path:

$$DTW(X, Y) = \sqrt{\sum_{(i,j)\in\pi} (X_i - Y_j)^2}$$

It is not required that both time series share the same size, but they must be the same dimension. DTW was originally presented in [1] and is discussed in more details in our *dedicated user-guide page*.

Parameters

s1 A time series.

s2 Another time series. If not given, self-similarity of dataset1 is returned.

global_constraint [{"itakura", "sakoe_chiba"} or None (default: None)] Global constraint to restrict admissible paths for DTW.

sakoe_chiba_radius [int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

itakura_max_slope [float or None (default: None)] Maximum slope for the Itakura parallelogram constraint. If None and *global_constraint* is set to "itakura", a maximum slope of 2. is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

Returns

list of integer pairs Matching path represented as a list of index pairs. In each pair, the first index corresponds to s1 and the second one corresponds to s2

float Similarity score

See also:

dtw Get only the similarity score for DTW

cdist_dtw Cross similarity matrix between time series datasets

dtw_path_from_metric Compute a DTW using a user-defined distance metric

References

[1]

Examples

```
>>> path, dist = dtw_path([1, 2, 3], [1., 2., 2., 3.])
>>> path
[(0, 0), (1, 1), (1, 2), (2, 3)]
>>> dist
0.0
>>> dtw_path([1, 2, 3], [1., 2., 2., 3., 4.])[1]
1.0
```

Examples using `tslearn.metrics.dtw_path`

- *DTW computation*

3.7.5 tslearn.metrics.dtw_path_from_metric

`tslearn.metrics.`**`dtw_path_from_metric`**(*s1*, *s2=None*, *metric='euclidean'*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, ***kwds*)

Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series using a distance metric defined by the user and return both the path and the similarity.

Similarity is computed as the cumulative cost along the aligned time series.

It is not required that both time series share the same size, but they must be the same dimension. DTW was originally presented in [1].

Valid values for metric are the same as for scikit-learn pairwise_distances function i.e. a string (e.g. "euclidean", "sqeuclidean", "hamming") or a function that is used to compute the pairwise distances. See scikit and scipy documentations for more information about the available metrics.

Parameters

s1 [array, shape = (sz1, d) if metric!="precomputed", (sz1, sz2) otherwise] A time series or an array of pairwise distances between samples.

s2 [array, shape = (sz2, d), optional (default: None)] A second time series, only allowed if metric != "precomputed".

metric [string or callable (default: "euclidean")] Function used to compute the pairwise distances between each points of *s1* and *s2*.

If metric is "precomputed", *s1* is assumed to be a distance matrix.

If metric is an other string, it must be one of the options compatible with sklearn.metrics.pairwise_distances.

Alternatively, if metric is a callable function, it is called on pairs of rows of *s1* and *s2*. The callable should take two 1 dimensional arrays as input and return a value indicating the distance between them.

global_constraint [{"itakura", "sakoe_chiba"} or None (default: None)] Global constraint to restrict admissible paths for DTW.

sakoe_chiba_radius [int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

itakura_max_slope [float or None (default: None)] Maximum slope for the Itakura parallelogram constraint. If None and *global_constraint* is set to "itakura", a maximum slope of 2. is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

****kwds** Additional arguments to pass to sklearn pairwise_distances to compute the pairwise distances.

Returns

list of integer pairs Matching path represented as a list of index pairs. In each pair, the first index corresponds to s1 and the second one corresponds to s2.

float Similarity score (sum of metric along the wrapped time series).

See also:

dtw_path Get both the matching path and the similarity score for DTW

Notes

By using a squared euclidean distance metric as shown above, the output path is the same as the one obtained by using dtw_path but the similarity score is the sum of squared distances instead of the euclidean distance.

References

[1]

Examples

Lets create 2 numpy arrays to wrap:

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> s1, s2 = rng.rand(5, 2), rng.rand(6, 2)
```

The wrapping can be done by passing a string indicating the metric to pass to scikit-learn pairwise_distances:

```
>>> dtw_path_from_metric(s1, s2,
...                      metric="sqeuclidean")  # doctest: +ELLIPSIS
([(0, 0), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5)], 1.117...)
```

Or by defining a custom distance function:

```
>>> sqeuclidean = lambda x, y: np.sum((x-y)**2)
>>> dtw_path_from_metric(s1, s2, metric=sqeuclidean)  # doctest: +ELLIPSIS
([(0, 0), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5)], 1.117...)
```

Or by using a precomputed distance matrix as input:

```
>>> from sklearn.metrics.pairwise import pairwise_distances
>>> dist_matrix = pairwise_distances(s1, s2, metric="sqeuclidean")
>>> dtw_path_from_metric(dist_matrix,
...                      metric="precomputed")  # doctest: +ELLIPSIS
([(0, 0), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5)], 1.117...)
```

Examples using `tslearn.metrics.dtw_path_from_metric`

- *DTW computation with a custom distance metric*

3.7.6 tslearn.metrics.dtw_limited_warping_length

`tslearn.metrics.`**`dtw_limited_warping_length`**(*s1*, *s2*, *max_length*)

Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series under an upper bound constraint on the resulting path length and return the similarity cost.

DTW is computed as the Euclidean distance between aligned time series, i.e., if π is the optimal alignment path:

$$DTW(X, Y) = \sqrt{\sum_{(i,j)\in\pi} \|X_i - Y_j\|^2}$$

Note that this formula is still valid for the multivariate case.

It is not required that both time series share the same size, but they must be the same dimension. DTW was originally presented in [1]. This constrained-length variant was introduced in [2]. Both bariants are discussed in more details in our *dedicated user-guide page*

Parameters

s1 A time series.

s2 Another time series.

max_length [int] Maximum allowed warping path length. If greater than len(s1) + len(s2), then it is equivalent to unconstrained DTW. If lower than max(len(s1), len(s2)), no path can be found and a ValueError is raised.

Returns

float Similarity score

See also:

dtw Get the similarity score for DTW

dtw_path_limited_warping_length Get both the warping path and the similarity score for DTW with limited warping path length

References

[1], [2]

Examples

```
>>> dtw_limited_warping_length([1, 2, 3], [1., 2., 2., 3.], 5)
0.0
>>> dtw_limited_warping_length([1, 2, 3], [1., 2., 2., 3., 4.], 5)
1.0
```

3.7.7 tslearn.metrics.dtw_path_limited_warping_length

tslearn.metrics.**dtw_path_limited_warping_length**(*s1*, *s2*, *max_length*)

Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series under an upper bound constraint on the resulting path length and return the path as well as the similarity cost.

DTW is computed as the Euclidean distance between aligned time series, i.e., if π is the optimal alignment path:

$$DTW(X, Y) = \sqrt{\sum_{(i,j)\in\pi} \|X_i - Y_j\|^2}$$

Note that this formula is still valid for the multivariate case.

It is not required that both time series share the same size, but they must be the same dimension. DTW was originally presented in [1]. This constrained-length variant was introduced in [2]. Both variants are discussed in more details in our *dedicated user-guide page*

Parameters

s1 A time series.

s2 Another time series.

max_length [int] Maximum allowed warping path length. If greater than len(s1) + len(s2), then it is equivalent to unconstrained DTW. If lower than max(len(s1), len(s2)), no path can be found and a ValueError is raised.

Returns

list of integer pairs Optimal path

float Similarity score

See also:

dtw_limited_warping_length Get the similarity score for DTW with limited warping path length

dtw_path Get both the matching path and the similarity score for DTW

References

[1], [2]

Examples

```
>>> path, cost = dtw_path_limited_warping_length([1, 2, 3],
...                                               [1., 2., 2., 3.], 5)
>>> cost
0.0
>>> path
[(0, 0), (1, 1), (1, 2), (2, 3)]
>>> path, cost = dtw_path_limited_warping_length([1, 2, 3],
...                                               [1., 2., 2., 3., 4.], 5)
>>> cost
1.0
>>> path
[(0, 0), (1, 1), (1, 2), (2, 3), (2, 4)]
```

3.7.8 tslearn.metrics.subsequence_path

tslearn.metrics.**subsequence_path**(*acc_cost_mat*, *idx_path_end*)

Compute the optimal path through a accumulated cost matrix given the endpoint of the sequence.

Parameters

acc_cost_mat: array, shape = (sz1, sz2) The accumulated cost matrix comparing subsequence from a longer sequence.

idx_path_end: int The end position of the matched subsequence in the longer sequence.

Returns

path: list of tuples of integer pairs Matching path represented as a list of index pairs. In each pair, the first index corresponds to *subseq* and the second one corresponds to *longseq*. The startpoint of the Path is $P_0 = (0, ?)$ and it ends at $P_L = (len(subseq) - 1, idx\_path\_end)$

See also:

dtw_subsequence_path Get the similarity score for DTW

subsequence_cost_matrix Calculate the required cost matrix

Examples

```
>>> acc_cost_mat = numpy.array([[1., 0., 0., 1., 4.],
...                             [5., 1., 1., 0., 1.]])
>>> # calculate the globally optimal path
>>> optimal_end_point = numpy.argmin(acc_cost_mat[-1, :])
>>> path = subsequence_path(acc_cost_mat, optimal_end_point)
>>> path
[(0, 2), (1, 3)]
```

Examples using tslearn.metrics.subsequence_path

- *sDTW multi path matching*

3.7.9 tslearn.metrics.subsequence_cost_matrix

tslearn.metrics.**subsequence_cost_matrix**(*subseq*, *longseq*)

Compute the accumulated cost matrix score between a subsequence and a reference time series.

Parameters

subseq [array, shape = (sz1, d)] Subsequence time series.

longseq [array, shape = (sz2, d)] Reference time series

Returns

mat [array, shape = (sz1, sz2)] Accumulated cost matrix.

Examples using tslearn.metrics.subsequence_cost_matrix

- *sDTW multi path matching*

3.7.10 tslearn.metrics.dtw_subsequence_path

tslearn.metrics.**dtw_subsequence_path**(*subseq*, *longseq*)

Compute sub-sequence Dynamic Time Warping (DTW) similarity measure between a (possibly multidimensional) query and a long time series and return both the path and the similarity.

DTW is computed as the Euclidean distance between aligned time series, i.e., if π is the alignment path:

$$DTW(X, Y) = \sqrt{\sum_{(i,j)\in\pi} \|X_i - Y_j\|^2}$$

Compared to traditional DTW, here, border constraints on admissible paths π are relaxed such that $\pi_0 = (0, ?)$ and $\pi_L = (N-1, ?)$ where L is the length of the considered path and N is the length of the subsequence time series.

It is not required that both time series share the same size, but they must be the same dimension. This implementation finds the best matching starting and ending positions for *subseq* inside *longseq*.

Parameters

subseq [array, shape = (sz1, d)] A query time series.

longseq [array, shape = (sz2, d)] A reference (supposed to be longer than *subseq*) time series.

Returns

list of integer pairs Matching path represented as a list of index pairs. In each pair, the first index corresponds to *subseq* and the second one corresponds to *longseq*.

float Similarity score

See also:

dtw Get the similarity score for DTW

subsequence_cost_matrix Calculate the required cost matrix

subsequence_path Calculate a matching path manually

Examples

```
>>> path, dist = dtw_subsequence_path([2., 3.], [1., 2., 2., 3., 4.])
>>> path
[(0, 2), (1, 3)]
>>> dist
0.0
```

3.7.11 tslearn.metrics.gak

tslearn.metrics.**gak**(*s1*, *s2*, *sigma=1.0*)

Compute Global Alignment Kernel (GAK) between (possibly multidimensional) time series and return it.

It is not required that both time series share the same size, but they must be the same dimension. GAK was originally presented in [1]. This is a normalized version that ensures that $k(x,x) = 1$ for all x and $k(x,y) \in [0, 1]$ for all x, y.

Parameters

s1 A time series

s2 Another time series

sigma [float (default 1.)] Bandwidth of the internal gaussian kernel used for GAK

Returns

float Kernel value

See also:

cdist_gak Compute cross-similarity matrix using Global Alignment kernel

References

[1]

Examples

```
>>> gak([1, 2, 3], [1., 2., 2., 3.], sigma=2.)  # doctest: +ELLIPSIS
0.839...
>>> gak([1, 2, 3], [1., 2., 2., 3., 4.])  # doctest: +ELLIPSIS
0.273...
```

3.7.12 tslearn.metrics.soft_dtw

tslearn.metrics.**soft_dtw**(*ts1*, *ts2*, *gamma=1.0*)

Compute Soft-DTW metric between two time series.

Soft-DTW was originally presented in [1] and is discussed in more details in our *user-guide page on DTW and its variants*.

Soft-DTW is computed as:

$$\text{soft-DTW}_\gamma(X, Y) = \min_\pi{}^\gamma \sum_{(i,j)\in\pi} \|X_i, Y_j\|^2$$

where $\min^\gamma$ is the soft-min operator of parameter γ.

In the limit case $\gamma = 0$, $\min^\gamma$ reduces to a hard-min operator and soft-DTW is defined as the square of the DTW similarity measure.

Parameters

ts1 A time series

ts2 Another time series

gamma [float (default 1.)] Gamma paraneter for Soft-DTW

Returns

float Similarity

See also:

cdist_soft_dtw Cross similarity matrix between time series datasets

References

[1]

Examples

```
>>> soft_dtw([1, 2, 2, 3],
...          [1., 2., 3., 4.],
...          gamma=1.)  # doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
-0.89...
>>> soft_dtw([1, 2, 3, 3],
...          [1., 2., 2.1, 3.2],
...          gamma=0.01)  # doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
0.089...
```

3.7.13 tslearn.metrics.cdist_soft_dtw

tslearn.metrics.**cdist_soft_dtw**(*dataset1*, *dataset2=None*, *gamma=1.0*)

Compute cross-similarity matrix using Soft-DTW metric.

Soft-DTW was originally presented in [1] and is discussed in more details in our *user-guide page on DTW and its variants*.

Soft-DTW is computed as:

$$\text{soft-DTW}_\gamma(X, Y) = \min_\pi{}^\gamma \sum_{(i,j)\in\pi} \|X_i, Y_j\|^2$$

where $\min^\gamma$ is the soft-min operator of parameter γ.

In the limit case $\gamma = 0$, $\min^\gamma$ reduces to a hard-min operator and soft-DTW is defined as the square of the DTW similarity measure.

Parameters

dataset1 A dataset of time series

dataset2 Another dataset of time series

gamma [float (default 1.)] Gamma paraneter for Soft-DTW

Returns

numpy.ndarray Cross-similarity matrix

See also:

soft_dtw Compute Soft-DTW

cdist_soft_dtw_normalized Cross similarity matrix between time series datasets using a normalized version of Soft-DTW

References

[1]

Examples

```
>>> cdist_soft_dtw([[1, 2, 2, 3], [1., 2., 3., 4.]], gamma=.01)
array([[-0.01098612,  1.        ],
       [ 1.        ,  0.        ]])
>>> cdist_soft_dtw([[1, 2, 2, 3], [1., 2., 3., 4.]],
...                [[1, 2, 2, 3], [1., 2., 3., 4.]], gamma=.01)
array([[-0.01098612,  1.        ],
       [ 1.        ,  0.        ]])
```

3.7.14 tslearn.metrics.cdist_soft_dtw_normalized

`tslearn.metrics.`**`cdist_soft_dtw_normalized`**(*dataset1*, *dataset2=None*, *gamma=1.0*)

Compute cross-similarity matrix using a normalized version of the Soft-DTW metric.

Soft-DTW was originally presented in [1] and is discussed in more details in our *user-guide page on DTW and its variants*.

Soft-DTW is computed as:

$$\text{soft-DTW}_\gamma(X, Y) = \min_\pi{}^\gamma \sum_{(i,j)\in\pi} \|X_i, Y_j\|^2$$

where $\min^\gamma$ is the soft-min operator of parameter γ.

In the limit case $\gamma = 0$, $\min^\gamma$ reduces to a hard-min operator and soft-DTW is defined as the square of the DTW similarity measure.

This normalized version is defined as:

$$\text{norm-soft-DTW}_\gamma(X, Y) = \text{soft-DTW}_\gamma(X, Y) - \frac{1}{2}\left(\text{soft-DTW}_\gamma(X, X) + \text{soft-DTW}_\gamma(Y, Y)\right)$$

and ensures that all returned values are positive and that $\text{norm-soft-DTW}_\gamma(X, X) = 0$.

Parameters

dataset1 A dataset of time series

dataset2 Another dataset of time series

gamma [float (default 1.)] Gamma paraneter for Soft-DTW

Returns

numpy.ndarray Cross-similarity matrix

See also:

soft_dtw Compute Soft-DTW

cdist_soft_dtw Cross similarity matrix between time series datasets using the unnormalized version of Soft-DTW

References

[1]

Examples

```
>>> time_series = numpy.random.randn(10, 15, 1)
>>> numpy.alltrue(cdist_soft_dtw_normalized(time_series) >= 0.)
True
```

3.7.15 tslearn.metrics.lb_envelope

`tslearn.metrics.`**`lb_envelope`**(*ts*, *radius=1*)

Compute time-series envelope as required by LB_Keogh.

LB_Keogh was originally presented in [1].

Parameters

ts [array-like] Time-series for which the envelope should be computed.

radius [int (default: 1)] Radius to be used for the envelope generation (the envelope at time index i will be generated based on all observations from the time series at indices comprised between i-radius and i+radius).

Returns

array-like Lower-side of the envelope.

array-like Upper-side of the envelope.

See also:

lb_keogh Compute LB_Keogh similarity

References

[1]

Examples

```
>>> ts1 = [1, 2, 3, 2, 1]
>>> env_low, env_up = lb_envelope(ts1, radius=1)
>>> env_low
array([[1.],
       [1.],
       [2.],
       [1.],
       [1.]])
>>> env_up
array([[2.],
       [3.],
       [3.],
       [3.],
       [2.]])
```

Examples using `tslearn.metrics.lb_envelope`

- *LB_Keogh*

3.7.16 tslearn.metrics.lb_keogh

`tslearn.metrics.`**`lb_keogh`**(*ts_query*, *ts_candidate=None*, *radius=1*, *envelope_candidate=None*)

Compute LB_Keogh.

LB_Keogh was originally presented in [1].

Parameters

ts_query [array-like] Query time-series to compare to the envelope of the candidate.

ts_candidate [array-like or None (default: None)] Candidate time-series. None means the envelope is provided via *envelope_candidate* parameter and hence does not need to be computed again.

radius [int (default: 1)] Radius to be used for the envelope generation (the envelope at time index i will be generated based on all observations from the candidate time series at indices comprised between i-radius and i+radius). Not used if *ts_candidate* is None.

envelope_candidate: pair of array-like (envelope_down, envelope_up) or None

(default: None) Pre-computed envelope of the candidate time series. If set to None, it is computed based on *ts_candidate*.

Returns

float Distance between the query time series and the envelope of the candidate time series.

See also:

lb_envelope Compute LB_Keogh-related envelope

Notes

This method requires a *ts_query* and *ts_candidate* (or *envelope_candidate*, depending on the call) to be of equal size.

References

[1]

Examples

```
>>> ts1 = [1, 2, 3, 2, 1]
>>> ts2 = [0, 0, 0, 0, 0]
>>> env_low, env_up = lb_envelope(ts1, radius=1)
>>> lb_keogh(ts_query=ts2,
...          envelope_candidate=(env_low, env_up))  # doctest: +ELLIPSIS
2.8284...
>>> lb_keogh(ts_query=ts2,
...          ts_candidate=ts1,
...          radius=1)  # doctest: +ELLIPSIS
2.8284...
```

Examples using `tslearn.metrics.lb_keogh`

- *LB_Keogh*

3.7.17 tslearn.metrics.sigma_gak

`tslearn.metrics.`**`sigma_gak`**(*dataset*, *n_samples=100*, *random_state=None*)

Compute sigma value to be used for GAK.

This method was originally presented in [1].

Parameters

dataset A dataset of time series

n_samples [int (default: 100)] Number of samples on which median distance should be estimated

random_state [integer or numpy.RandomState or None (default: None)] The generator used to draw the samples. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Returns

float Suggested bandwidth (σ) for the Global Alignment kernel

See also:

gak Compute Global Alignment kernel

cdist_gak Compute cross-similarity matrix using Global Alignment kernel

References

[1]

Examples

```
>>> dataset = [[1, 2, 2, 3], [1., 2., 3., 4.]]
>>> sigma_gak(dataset=dataset,
...           n_samples=200,
...           random_state=0)  # doctest: +ELLIPSIS
2.0...
```

3.7.18 tslearn.metrics.gamma_soft_dtw

tslearn.metrics.**gamma_soft_dtw**(*dataset*, *n_samples=100*, *random_state=None*)

Compute gamma value to be used for GAK/Soft-DTW.

This method was originally presented in [1].

Parameters

dataset A dataset of time series

n_samples [int (default: 100)] Number of samples on which median distance should be estimated

random_state [integer or numpy.RandomState or None (default: None)] The generator used to draw the samples. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Returns

float Suggested γ parameter for the Soft-DTW

See also:

sigma_gak Compute sigma parameter for Global Alignment kernel

References

[1]

Examples

```
>>> dataset = [[1, 2, 2, 3], [1., 2., 3., 4.]]
>>> gamma_soft_dtw(dataset=dataset,
...                n_samples=200,
...                random_state=0)  # doctest: +ELLIPSIS
8.0...
```

3.8 tslearn.neural_network

The *tslearn.neural_network* module contains multi-layer perceptron models for time series classification and regression.

These are straight-forward adaptations of scikit-learn models.

Classes

TimeSeriesMLPClassifier([...])	A Multi-Layer Perceptron classifier for time series.
TimeSeriesMLPRegressor([hidden_layer_sizes, ...])	A Multi-Layer Perceptron regressor for time series.

3.8.1 tslearn.neural_network.TimeSeriesMLPClassifier

class tslearn.neural_network.**TimeSeriesMLPClassifier**(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*)

A Multi-Layer Perceptron classifier for time series.

This class mainly reshapes data so that it can be fed to *scikit-learn*'s `MLPClassifier`.

It accepts the exact same hyper-parameters as `MLPClassifier`, check scikit-learn docs for a list of parameters and attributes.

Attributes

partial_fit Update the model with a single iteration over the given data.

Notes

This method requires a dataset of equal-sized time series.

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=30, sz=16, d=2, n_blobs=3,
...                          random_state=0)
>>> mlp = TimeSeriesMLPClassifier(hidden_layer_sizes=(64, 64),
...                               random_state=0)
>>> mlp.fit(X, y)  # doctest: +ELLIPSIS
TimeSeriesMLPClassifier(...)
>>> [c.shape for c in mlp.coefs_]
[(32, 64), (64, 64), (64, 3)]
>>> [c.shape for c in mlp.intercepts_]
[(64,), (64,), (3,)]
```

Methods

fit(X, y)	Fit the model using X as training data and y as target values
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the class labels for the provided data
predict_log_proba(X)	Predict the class log-probabilities for the provided data
predict_proba(X)	Predict the class probabilities for the provided data
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

fit (*X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [array-like, shape (n_ts, sz, d)] Training data.

y [array-like, shape (n_ts,) or (n_ts, dim_y)] Target values.

Returns

TimeSeriesMLPClassifier The fitted estimator

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit
Update the model with a single iteration over the given data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

y [array-like, shape (n_samples,)] The target values.

classes [array, shape (n_classes), default None] Classes across all calls to partial_fit. Can be obtained via *np.unique(y_all)*, where y_all is the target vector of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in *classes*.

Returns

self [returns a trained MLP model.]

predict (*X*)
Predict the class labels for the provided data

Parameters

X [array-like, shape (n_ts, sz, d)] Test samples.

Returns

array, shape = (n_ts,) Array of predicted class labels

predict_log_proba(*X*)

Predict the class log-probabilities for the provided data

Parameters

X [array-like, shape (n_ts, sz, d)] Test samples.

Returns

array, shape = (n_ts, n_classes) Array of predicted class log-probabilities

predict_proba(*X*)

Predict the class probabilities for the provided data

Parameters

X [array-like, shape (n_ts, sz, d)] Test samples.

Returns

array, shape = (n_ts, n_classes) Array of predicted class probabilities

score(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

3.8.2 tslearn.neural_network.TimeSeriesMLPRegressor

class tslearn.neural_network.**TimeSeriesMLPRegressor**(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*)

A Multi-Layer Perceptron regressor for time series.

This class mainly reshapes data so that it can be fed to *scikit-learn*'s `MLPRegressor`.

It accepts the exact same hyper-parameters as `MLPRegressor`, check scikit-learn docs for a list of parameters and attributes.

Attributes

partial_fit Update the model with a single iteration over the given data.

Notes

This method requires a dataset of equal-sized time series.

Examples

```
>>> mlp = TimeSeriesMLPRegressor(hidden_layer_sizes=(64, 64),
...                              random_state=0)
>>> mlp.fit(X=[[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0, 0, 1])  # doctest: +ELLIPSIS
TimeSeriesMLPRegressor(...)
>>> [c.shape for c in mlp.coefs_]
[(3, 64), (64, 64), (64, 1)]
>>> [c.shape for c in mlp.intercepts_]
[(64,), (64,), (1,)]
```

Methods

fit(X, y)	Fit the model using X as training data and y as target values
get_params([deep])	Get parameters for this estimator.

Continued on next page

Table 19 – continued from previous page

predict(X)	Predict the target for the provided data
score(X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

fit (*X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [array-like, shape (n_ts, sz, d)] Training data.

y [array-like, shape (n_ts,) or (n_ts, dim_y)] Target values.

Returns

TimeSeriesMLPRegressor The fitted estimator

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit
Update the model with a single iteration over the given data.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input data.

y [ndarray of shape (n_samples,)] The target values.

Returns

self [returns a trained MLP model.]

predict (*X*)
Predict the target for the provided data

Parameters

X [array-like, shape (n_ts, sz, d)] Test samples.

Returns

array, shape = (n_ts,) or (n_ts, dim_y) Array of predicted targets

score (*X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

3.9 tslearn.neighbors

The *tslearn.neighbors* module gathers nearest neighbor algorithms using time series metrics.

Classes

KNeighborsTimeSeries([n_neighbors, metric, ...])	Unsupervised learner for implementing neighbor searches for Time Series.
KNeighborsTimeSeriesClassifier([...])	Classifier implementing the k-nearest neighbors vote for Time Series.
KNeighborsTimeSeriesRegressor([n_neighbors, ...])	Classifier implementing the k-nearest neighbors vote for Time Series.

3.9.1 tslearn.neighbors.KNeighborsTimeSeries

class tslearn.neighbors.**KNeighborsTimeSeries**(*n_neighbors=5*, *metric='dtw'*, *metric_params=None*, *n_jobs=None*, *verbose=0*)
Unsupervised learner for implementing neighbor searches for Time Series.

Parameters

n_neighbors [int (default: 5)] Number of nearest neighbors to be considered for the decision.

metric [{'dtw', 'softdtw', 'euclidean', 'sqeuclidean', 'cityblock', 'sax'} (default: 'dtw')] Metric to be used at the core of the nearest neighbor procedure. DTW and SAX are described in more detail in *tslearn.metrics*. When SAX is provided as a metric, the data is expected to be normalized such that each time series has zero mean and unit variance. Other metrics are described in scipy.spatial.distance doc.

metric_params [dict or None (default: None)] Dictionary of metric parameters. For metrics that accept parallelization of the cross-distance matrix computations, *n_jobs* and *verbose* keys passed in *metric_params* are overridden by the *n_jobs* and *verbose* arguments. For 'sax' metric, these are hyper-parameters to be passed at the creation of the *SymbolicAggregateApproximation* object.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for cross-distance matrix computations. Ignored if the cross-distance matrix cannot be computed using parallelization. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

Notes

The training data are saved to disk if this model is serialized and may result in a large model file if the training dataset is large.

Examples

```
>>> time_series = to_time_series_dataset([[1, 2, 3, 4],
...                                       [3, 3, 2, 0],
...                                       [1, 2, 2, 4]])
>>> knn = KNeighborsTimeSeries(n_neighbors=1).fit(time_series)
>>> dataset = to_time_series_dataset([[1, 1, 2, 2, 2, 3, 4]])
>>> dist, ind = knn.kneighbors(dataset, return_distance=True)
>>> dist
array([[0.]])
>>> print(ind)
[[0]]
>>> knn2 = KNeighborsTimeSeries(n_neighbors=10,
...                             metric="euclidean").fit(time_series)
>>> print(knn2.kneighbors(return_distance=False))
[[2 1]
 [2 0]
 [0 1]]
```

Methods

fit(X[, y])	Fit the model using X as training data
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.

Continued on next page

Table 21 – continued from previous page

kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
radius_neighbors([X, radius, ...])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph([X, radius, mode, ...])	Computes the (weighted) graph of Neighbors for points in X
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.

fit(*X*, *y=None*)
Fit the model using X as training data

Parameters

X [array-like, shape (n_ts, sz, d)] Training data.

classmethod from_hdf5(*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

kneighbors (*X=None*, *n_neighbors=None*, *return_distance=True*)
Finds the K-neighbors of a point.

Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_ts, sz, d)] The query time series. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the distance to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

kneighbors_graph (*X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

radius_neighbors(*X=None*, *radius=None*, *return_distance=True*, *sort_results=False*)

Finds the neighbors within a given radius of a point or points.

Return the indices and distances of each point from the dataset lying in a ball with size `radius` around the points of the query array. Points lying on the boundary are included in the results.

The result points are *not* necessarily sorted by distance to their query point.

Parameters

X [array-like, (n_samples, n_features), optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Limiting distance of neighbors to return. (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return_distance == False, setting sort_results = True will result in an error.

New in version 0.22.

Returns

neigh_dist [array, shape (n_samples,) of arrays] Array representing the distances to each point, only present if return_distance=True. The distance values are computed according to the `metric` constructor parameter.

neigh_ind [array, shape (n_samples,) of arrays] An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, *radius_neighbors* returns arrays of objects, where each object is a 1D array of indices or distances.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1, 1, 1]:

```
>>> import numpy as np
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.6)
>>> neigh.fit(samples)
NearestNeighbors(radius=1.6)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0]))
[1.5 0.5]
>>> print(np.asarray(rng[1][0]))
[1 2]
```

The first array returned contains the distances to all points which are closer than 1.6, while the second array returned contains their indices. In general, multiple points can be queried at the same time.

radius_neighbors_graph(*X=None*, *radius=None*, *mode='connectivity'*, *sort_results=False*)
Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Parameters

X [array-like of shape (n_samples, n_features), default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Radius of neighborhoods. (default is the value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

sort_results [boolean, optional. Defaults to False.] If True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. Only used with mode='distance'.

New in version 0.22.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(radius=1.5)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

Examples using `tslearn.neighbors.KNeighborsTimeSeries`

- *k-NN search*
- *Nearest neighbors*

3.9.2 tslearn.neighbors.KNeighborsTimeSeriesClassifier

class tslearn.neighbors.**KNeighborsTimeSeriesClassifier**(*n_neighbors=5*, *weights='uniform'*, *metric='dtw'*, *metric_params=None*, *n_jobs=None*, *verbose=0*)
Classifier implementing the k-nearest neighbors vote for Time Series.

Parameters

n_neighbors [int (default: 5)] Number of nearest neighbors to be considered for the decision.

weights [str or callable, optional (default: 'uniform')] Weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

metric [one of the metrics allowed for *KNeighborsTimeSeries*]

class (default: 'dtw') Metric to be used at the core of the nearest neighbor procedure

metric_params [dict or None (default: None)] Dictionnary of metric parameters. For metrics that accept parallelization of the cross-distance matrix computations, *n_jobs* and *verbose* keys passed in *metric_params* are overridden by the *n_jobs* and *verbose* arguments. For 'sax' metric, these are hyper-parameters to be passed at the creation of the *SymbolicAggregateApproximation* object.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for cross-distance matrix computations. Ignored if the cross-distance matrix cannot be computed using parallelization. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose [int, optional (default=0)] The verbosity level: if non zero, progress messages are printed. Above 50, the output is sent to stdout. The frequency of the messages increases with the verbosity level. If it more than 10, all iterations are reported. Glossary for more details.

Notes

The training data are saved to disk if this model is serialized and may result in a large model file if the training dataset is large.

Examples

```
>>> clf = KNeighborsTimeSeriesClassifier(n_neighbors=2, metric="dtw")
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0, 0, 1]).predict([[1, 2.2, 3.5]])
array([0])
>>> clf = KNeighborsTimeSeriesClassifier(n_neighbors=2,
...                                      metric="dtw",
...                                      n_jobs=2)
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0, 0, 1]).predict([[1, 2.2, 3.5]])
array([0])
>>> clf = KNeighborsTimeSeriesClassifier(n_neighbors=2,
...                                      metric="dtw",
...                                      metric_params={
...                                          "itakura_max_slope": 2.},
...                                      n_jobs=2)
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0, 0, 1]).predict([[1, 2.2, 3.5]])
array([0])
```

Methods

fit(X, y)	Fit the model using X as training data and y as target values
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.

Continued on next page

Table 22 – continued from previous page

kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(X)	Predict the class labels for the provided data
predict_proba(X)	Predict the class probabilities for the provided data
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.

fit(*X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [array-like, shape (n_ts, sz, d)] Training data.

y [array-like, shape (n_ts,)] Target values.

Returns

KNeighborsTimeSeriesClassifier The fitted estimator

classmethod from_hdf5(*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

kneighbors (*X=None*, *n_neighbors=None*, *return_distance=True*)
Finds the K-neighbors of a point.

Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_ts, sz, d)] The query time series. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the distance to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

kneighbors_graph (*X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
```

(continues on next page)

(continued from previous page)

```
       [0., 1., 1.],
       [1., 0., 1.]])
```

predict(*X*)
Predict the class labels for the provided data

Parameters

X [array-like, shape (n_ts, sz, d)] Test samples.

Returns

array, shape = (n_ts,) Array of predicted class labels

predict_proba(*X*)
Predict the class probabilities for the provided data

Parameters

X [array-like, shape (n_ts, sz, d)] Test samples.

Returns

array, shape = (n_ts, n_classes) Array of predicted class probabilities

score(*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

Examples using `tslearn.neighbors.KNeighborsTimeSeriesClassifier`

- *Nearest neighbors*
- *Hyper-parameter tuning of a Pipeline with KNeighborsTimeSeriesClassifier*
- *1-NN with SAX + MINDIST*

3.9.3 tslearn.neighbors.KNeighborsTimeSeriesRegressor

class tslearn.neighbors.**KNeighborsTimeSeriesRegressor**(*n_neighbors=5*, *weights='uniform'*, *metric='dtw'*, *metric_params=None*, *n_jobs=None*, *verbose=0*)
Classifier implementing the k-nearest neighbors vote for Time Series.

Parameters

n_neighbors [int (default: 5)] Number of nearest neighbors to be considered for the decision.

weights [str or callable, optional (default: 'uniform')] Weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

metric [one of the metrics allowed for *KNeighborsTimeSeries*]

class (default: 'dtw') Metric to be used at the core of the nearest neighbor procedure

metric_params [dict or None (default: None)] Dictionnary of metric parameters. For metrics that accept parallelization of the cross-distance matrix computations, *n_jobs* and *verbose* keys passed in *metric_params* are overridden by the *n_jobs* and *verbose* arguments. For 'sax' metric, these are hyper-parameters to be passed at the creation of the *SymbolicAggregateApproximation* object.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for cross-distance matrix computations. Ignored if the cross-distance matrix cannot be computed using parallelization. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose [int, optional (default=0)] The verbosity level: if non zero, progress messages are printed. Above 50, the output is sent to stdout. The frequency of the messages increases with the verbosity level. If it more than 10, all iterations are reported. Glossary for more details.

Examples

```
>>> clf = KNeighborsTimeSeriesRegressor(n_neighbors=2, metric="dtw")
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0.1, 0.1, 1.1]).predict([[1, 2.2, 3.5]])
array([0.1])
>>> clf = KNeighborsTimeSeriesRegressor(n_neighbors=2,
...                                     metric="dtw",
...                                     n_jobs=2)
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0.1, 0.1, 1.1]).predict([[1, 2.2, 3.5]])
array([0.1])
>>> clf = KNeighborsTimeSeriesRegressor(n_neighbors=2,
...                                     metric="dtw",
...                                     metric_params={
...                                         "itakura_max_slope": 2.},
...                                     n_jobs=2)
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0.1, 0.1, 1.1]).predict([[1, 2.2, 3.5]])
array([0.1])
```

Methods

fit(X, y)	Fit the model using X as training data and y as target values
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(X)	Predict the target for the provided data
score(X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

fit (*X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [array-like, shape (n_ts, sz, d)] Training data.

y [array-like, shape (n_ts,) or (n_ts, dim_y)] Target values.

Returns

KNeighborsTimeSeriesRegressor The fitted estimator

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

kneighbors (*X=None*, *n_neighbors=None*, *return_distance=True*)
Finds the K-neighbors of a point.

Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_ts, sz, d)] The query time series. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the distance to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

kneighbors_graph (*X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
```

(continues on next page)

(continued from previous page)

```
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

predict(*X*)

Predict the target for the provided data

Parameters

X [array-like, shape (n_ts, sz, d)] Test samples.

Returns

array, shape = (n_ts,) or (n_ts, dim_y) Array of predicted targets

score(*X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

3.10 tslearn.piecewise

The *tslearn.piecewise* module gathers time series piecewise approximation algorithms.

Classes

OneD_SymbolicAggregateApproximation([...])	One-D Symbolic Aggregate approXimation (1d-SAX) transformation.
PiecewiseAggregateApproximation([n_segments])	Piecewise Aggregate Approximation (PAA) transformation.
SymbolicAggregateApproximation([n_segments, ...])	Symbolic Aggregate approXimation (SAX) transformation.

3.10.1 tslearn.piecewise.OneD_SymbolicAggregateApproximation

class tslearn.piecewise.**OneD_SymbolicAggregateApproximation**(*n_segments=1*, *alphabet_size_avg=5*, *alphabet_size_slope=5*, *sigma_l=None*, *scale=False*)

One-D Symbolic Aggregate approXimation (1d-SAX) transformation.

1d-SAX was originally presented in [1].

Parameters

n_segments [int (default: 1)] Number of PAA segments to compute.

alphabet_size_avg [int (default: 5)] Number of SAX symbols to use to describe average values.

alphabet_size_slope [int (default: 5)] Number of SAX symbols to use to describe slopes.

sigma_l [float or None (default: None)] Scale parameter of the Gaussian distribution used to quantize slopes. If None, the formula given in [1] is used: $\sigma_L = \sqrt{0.03/L}$ where L is the length of each segment.

scale: bool (default: False) Whether input data should be scaled for each feature of each time series to have zero mean and unit variance. Default for this parameter is set to *False* in version 0.4 to ensure backward compatibility, but is likely to change in a future version.

Attributes

breakpoints_avg_ [numpy.ndarray of shape (alphabet_size_avg - 1,)] List of breakpoints used to generate SAX symbols for average values.

breakpoints_slope_ [numpy.ndarray of shape (alphabet_size_slope - 1,)] List of breakpoints used to generate SAX symbols for slopes.

Notes

This method requires a dataset of equal-sized time series.

References

[1]

Examples

```
>>> one_d_sax = OneD_SymbolicAggregateApproximation(n_segments=3,
...         alphabet_size_avg=2, alphabet_size_slope=2, sigma_l=1.)
>>> data = [[-1., 2., 0.1, -1., 1., -1.], [1., 3.2, -1., -3., 1., -1.]]
>>> one_d_sax_data = one_d_sax.fit_transform(data)
>>> one_d_sax_data.shape
(2, 3, 2)
>>> one_d_sax_data
array([[[1, 1],
        [0, 0],
        [1, 0]],
<BLANKLINE>
       [[1, 1],
        [0, 0],
        [1, 0]]])
>>> one_d_sax.distance_sax(one_d_sax_data[0], one_d_sax_data[1])
0.0
>>> one_d_sax.distance(data[0], data[1])
0.0
>>> one_d_sax.inverse_transform(one_d_sax_data)
array([[[ 0.33724488],
        [ 1.01173463],
        [-0.33724488],
        [-1.01173463],
        [ 1.01173463],
        [ 0.33724488]],
<BLANKLINE>
       [[ 0.33724488],
        [ 1.01173463],
        [-0.33724488],
        [-1.01173463],
        [ 1.01173463],
        [ 0.33724488]]])
>>> one_d_sax.fit(data).sigma_l
1.0
```

Methods

distance(ts1, ts2)	Compute distance between 1d-SAX representations as defined in [1].
distance_1d_sax(sax1, sax2)	Compute distance between 1d-SAX representations as defined in [1].
distance_paa(paa1, paa2)	Compute distance between PAA representations as defined in [1].
distance_sax(sax1, sax2)	Compute distance between SAX representations as defined in [1].
fit(X[, y])	Fit a 1d-SAX representation.

Continued on next page

Table 25 – continued from previous page

fit_transform(X[, y])	Fit a 1d-SAX representation and transform the data accordingly.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Compute time series corresponding to given 1d-SAX representations.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X[, y])	Transform a dataset of time series into its 1d-SAX representation.

distance(*ts1*, *ts2*)
Compute distance between 1d-SAX representations as defined in [1].

Parameters

ts1 [array-like] A time series

ts2 [array-like] Another time series

Returns

float 1d-SAX distance

References

[1]

distance_1d_sax(*sax1*, *sax2*)
Compute distance between 1d-SAX representations as defined in [1].

Parameters

sax1 [array-like] 1d-SAX representation of a time series

sax2 [array-like] 1d-SAX representation of another time series

Returns

float 1d-SAX distance

Notes

Unlike SAX distance, 1d-SAX distance does not lower bound Euclidean distance between original time series.

References

[1]

distance_paa(*paa1*, *paa2*)
Compute distance between PAA representations as defined in [1].

Parameters

paa1 [array-like] PAA representation of a time series

paa2 [array-like] PAA representation of another time series

Returns

float PAA distance

References

[1]

distance_sax(*sax1*, *sax2*)
Compute distance between SAX representations as defined in [1].

Parameters

sax1 [array-like] SAX representation of a time series

sax2 [array-like] SAX representation of another time series

Returns

float SAX distance

References

[1]

fit(*X*, *y=None*)
Fit a 1d-SAX representation.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

OneD_SymbolicAggregateApproximation self

fit_transform(*X*, *y=None*, ***fit_params*)
Fit a 1d-SAX representation and transform the data accordingly.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

numpy.ndarray of integers with shape (n_ts, n_segments, 2 * d) 1d-SAX-Transformed dataset. The order of the last dimension is: first d elements represent average values (standard SAX symbols) and the last d are for slopes

classmethod from_hdf5(*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*X*)
Compute time series corresponding to given 1d-SAX representations.

Parameters

X [array-like of shape (n_ts, sz_sax, 2 * d)] A dataset of SAX series.

Returns

numpy.ndarray of shape (n_ts, sz_original_ts, d) A dataset of time series corresponding to the provided representation.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

transform(*X*, *y=None*)
Transform a dataset of time series into its 1d-SAX representation.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

numpy.ndarray of integers with shape (n_ts, n_segments, 2 * d) 1d-SAX-Transformed dataset

Examples using `tslearn.piecewise.OneD_SymbolicAggregateApproximation`

- *PAA and SAX features*

3.10.2 tslearn.piecewise.PiecewiseAggregateApproximation

class tslearn.piecewise.**PiecewiseAggregateApproximation**(*n_segments=1*)
Piecewise Aggregate Approximation (PAA) transformation.

PAA was originally presented in [1].

Parameters

n_segments [int (default: 1)] Number of PAA segments to compute

Notes

This method requires a dataset of equal-sized time series.

References

[1]

Examples

```
>>> paa = PiecewiseAggregateApproximation(n_segments=3)
>>> data = [[-1., 2., 0.1, -1., 1., -1.], [1., 3.2, -1., -3., 1., -1.]]
>>> paa_data = paa.fit_transform(data)
>>> paa_data.shape
(2, 3, 1)
>>> paa_data
array([[[ 0.5 ],
        [-0.45],
        [ 0.  ]],
<BLANKLINE>
       [[ 2.1 ],
        [-2.  ],
        [ 0.  ]]])
>>> paa.distance_paa(paa_data[0], paa_data[1])  # doctest: +ELLIPSIS
3.15039...
>>> paa.distance(data[0], data[1])  # doctest: +ELLIPSIS
3.15039...
>>> paa.inverse_transform(paa_data)
array([[[ 0.5 ],
        [ 0.5 ],
        [-0.45],
        [-0.45],
        [ 0.  ],
        [ 0.  ]],
<BLANKLINE>
       [[ 2.1 ],
        [ 2.1 ],
        [-2.  ],
        [-2.  ],
        [ 0.  ],
        [ 0.  ]]])
```

Methods

distance(ts1, ts2)	Compute distance between PAA representations as defined in [1].
distance_paa(paa1, paa2)	Compute distance between PAA representations as defined in [1].
fit(X[, y])	Fit a PAA representation.
fit_transform(X[, y])	Fit a PAA representation and transform the data accordingly.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Compute time series corresponding to given PAA representations.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X[, y])	Transform a dataset of time series into its PAA representation.

distance(*ts1*, *ts2*)
Compute distance between PAA representations as defined in [1].

Parameters

ts1 [array-like] A time series

ts2 [array-like] Another time series

Returns

float PAA distance

References

[1]

distance_paa(*paa1*, *paa2*)
Compute distance between PAA representations as defined in [1].

Parameters

paa1 [array-like] PAA representation of a time series

paa2 [array-like] PAA representation of another time series

Returns

float PAA distance

References

[1]

fit(*X*, *y=None*)
Fit a PAA representation.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

PiecewiseAggregateApproximation self

fit_transform(*X*, *y=None*, ***fit_params*)
Fit a PAA representation and transform the data accordingly.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

numpy.ndarray of shape (n_ts, n_segments, d) PAA-Transformed dataset

classmethod from_hdf5(*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*X*)
Compute time series corresponding to given PAA representations.

Parameters

X [array-like of shape (n_ts, sz_paa, d)] A dataset of PAA series.

Returns

numpy.ndarray of shape (n_ts, sz_original_ts, d) A dataset of time series corresponding to the provided representation.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

transform(*X*, *y=None*)
Transform a dataset of time series into its PAA representation.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

numpy.ndarray of shape (n_ts, n_segments, d) PAA-Transformed dataset

Examples using `tslearn.piecewise.PiecewiseAggregateApproximation`

- *PAA and SAX features*

3.10.3 tslearn.piecewise.SymbolicAggregateApproximation

class tslearn.piecewise.**SymbolicAggregateApproximation**(*n_segments=1*, *alphabet_size_avg=5*, *scale=False*)

Symbolic Aggregate approXimation (SAX) transformation.

SAX was originally presented in [1].

Parameters

n_segments [int (default: 1)] Number of PAA segments to compute

alphabet_size_avg [int (default: 5)] Number of SAX symbols to use

scale: bool (default: False) Whether input data should be scaled for each feature to have zero mean and unit variance across the dataset passed at fit time. Default for this parameter is set to *False* in version 0.4 to ensure backward compatibility, but is likely to change in a future version.

Attributes

breakpoints_avg_ [numpy.ndarray of shape (alphabet_size - 1,)] List of breakpoints used to generate SAX symbols

Notes

This method requires a dataset of equal-sized time series.

References

[1]

Examples

```
>>> sax = SymbolicAggregateApproximation(n_segments=3, alphabet_size_avg=2)
>>> data = [[-1., 2., 0.1, -1., 1., -1.], [1., 3.2, -1., -3., 1., -1.]]
>>> sax_data = sax.fit_transform(data)
>>> sax_data.shape
(2, 3, 1)
>>> sax_data
array([[[1],
        [0],
        [1]],
<BLANKLINE>
       [[1],
        [0],
        [1]]])
>>> sax.distance_sax(sax_data[0], sax_data[1])  # doctest: +ELLIPSIS
0.0
>>> sax.distance(data[0], data[1])  # doctest: +ELLIPSIS
0.0
>>> sax.inverse_transform(sax_data)
array([[[ 0.67448975],
        [ 0.67448975],
        [-0.67448975],
        [-0.67448975],
        [ 0.67448975],
        [ 0.67448975]],
<BLANKLINE>
       [[ 0.67448975],
        [ 0.67448975],
        [-0.67448975],
        [-0.67448975],
        [ 0.67448975],
        [ 0.67448975]]])
```

Methods

distance(ts1, ts2)	Compute distance between SAX representations as defined in [1].
distance_paa(paa1, paa2)	Compute distance between PAA representations as defined in [1].
distance_sax(sax1, sax2)	Compute distance between SAX representations as defined in [1].
fit(X[, y])	Fit a SAX representation.
fit_transform(X[, y])	Fit a SAX representation and transform the data accordingly.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.

Continued on next page

Table 27 – continued from previous page

get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Compute time series corresponding to given SAX representations.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X[, y])	Transform a dataset of time series into its SAX representation.

distance(*ts1*, *ts2*)
Compute distance between SAX representations as defined in [1].

Parameters

ts1 [array-like] A time series

ts2 [array-like] Another time series

Returns

float SAX distance

References

[1]

distance_paa(*paa1*, *paa2*)
Compute distance between PAA representations as defined in [1].

Parameters

paa1 [array-like] PAA representation of a time series

paa2 [array-like] PAA representation of another time series

Returns

float PAA distance

References

[1]

distance_sax(*sax1*, *sax2*)
Compute distance between SAX representations as defined in [1].

Parameters

sax1 [array-like] SAX representation of a time series

sax2 [array-like] SAX representation of another time series

Returns

float SAX distance

References

[1]

fit (*X*, *y=None*)
Fit a SAX representation.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

SymbolicAggregateApproximation self

fit_transform (*X*, *y=None*, ***fit_params*)
Fit a SAX representation and transform the data accordingly.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

numpy.ndarray of integers with shape (n_ts, n_segments, d) SAX-Transformed dataset

classmethod from_hdf5 (*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json (*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle (*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*X*)
Compute time series corresponding to given SAX representations.

Parameters

X [array-like of shape (n_ts, sz_sax, d)] A dataset of SAX series.

Returns

numpy.ndarray of shape (n_ts, sz_original_ts, d) A dataset of time series corresponding to the provided representation.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

transform(*X*, *y=None*)
Transform a dataset of time series into its SAX representation.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset

Returns

numpy.ndarray of integers with shape (n_ts, n_segments, d) SAX-Transformed dataset

Examples using `tslearn.piecewise.SymbolicAggregateApproximation`

- *PAA and SAX features*

3.11 tslearn.preprocessing

The *tslearn.preprocessing* module gathers time series scalers and resamplers.

Classes

TimeSeriesScalerMeanVariance([mu, std])	Scaler for time series.
TimeSeriesScalerMinMax([value_range])	Scaler for time series.
TimeSeriesResampler(sz)	Resampler for time series.

3.11.1 tslearn.preprocessing.TimeSeriesScalerMeanVariance

class tslearn.preprocessing.**TimeSeriesScalerMeanVariance**(*mu=0.0*, *std=1.0*)

Scaler for time series. Scales time series so that their mean (resp. standard deviation) in each dimension is mu (resp. std).

Parameters

mu [float (default: 0.)] Mean of the output time series.

std [float (default: 1.)] Standard deviation of the output time series.

Notes

This method requires a dataset of equal-sized time series.

NaNs within a time series are ignored when calculating mu and std.

Examples

```
>>> TimeSeriesScalerMeanVariance(mu=0.,
...                              std=1.).fit_transform([[0, 3, 6]])
array([[[-1.22474487],
        [ 0.        ],
        [ 1.22474487]]])
>>> TimeSeriesScalerMeanVariance(mu=0.,
...                              std=1.).fit_transform([[numpy.nan, 3, 6]])
array([[[nan],
        [-1.],
        [ 1.]]])
```

Methods

fit(X[, y])	A dummy method such that it complies to the sklearn requirements.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Fit to data, then transform it.

fit(*X*, *y=None*, ***kwargs*)
A dummy method such that it complies to the sklearn requirements. Since this method is completely stateless, it just returns itself.

Parameters

X Ignored

Returns

self

fit_transform(*X*, *y=None*, ***kwargs*)
Fit to data, then transform it.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset to be rescaled.

Returns

numpy.ndarray Resampled time series dataset.

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform(*X*, *y=None*, ***kwargs*)
Fit to data, then transform it.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset to be rescaled

Returns

numpy.ndarray Rescaled time series dataset

Examples using `tslearn.preprocessing.TimeSeriesScalerMeanVariance`

- *LB_Keogh*
- *sDTW multi path matching*
- *DTW computation with a custom distance metric*
- *1-NN with SAX + MINDIST*
- *KShape*
- *Kernel k-means*
- *k-means*
- *Early Classification*
- *Model Persistence*
- *PAA and SAX features*
- *Distance and Matrix Profiles*

3.11.2 `tslearn.preprocessing`.TimeSeriesScalerMinMax

class tslearn.preprocessing.**TimeSeriesScalerMinMax**(*value_range=(0.0, 1.0)*)

Scaler for time series. Scales time series so that their span in each dimension is between `min` and `max` where `value_range=(min, max)`.

Parameters

value_range [tuple (default: (0., 1.))] The minimum and maximum value for the output time series.

Notes

This method requires a dataset of equal-sized time series.

NaNs within a time series are ignored when calculating min and max.

Examples

```
>>> TimeSeriesScalerMinMax(value_range=(1., 2.)).fit_transform([[0, 3, 6]])
array([[[1. ],
        [1.5],
        [2. ]]])
>>> TimeSeriesScalerMinMax(value_range=(1., 2.)).fit_transform(
...     [[numpy.nan, 3, 6]]
... )
array([[[nan],
        [ 1.],
        [ 2.]]])
```

Methods

fit(X[, y])	A dummy method such that it complies to the sklearn requirements.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Will normalize (min-max) each of the timeseries.

fit (*X*, *y=None*, ***kwargs*)
A dummy method such that it complies to the sklearn requirements. Since this method is completely stateless, it just returns itself.

Parameters

X Ignored

Returns

self

fit_transform (*X*, *y=None*, ***kwargs*)
Fit to data, then transform it.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset to be rescaled.

Returns

numpy.ndarray Resampled time series dataset.

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

transform (*X*, *y=None*, ***kwargs*)
Will normalize (min-max) each of the timeseries. IMPORTANT: this transformation is completely stateless, and is applied to each of the timeseries individually.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset to be rescaled.

Returns

numpy.ndarray Rescaled time series dataset.

Examples using `tslearn.preprocessing.TimeSeriesScalerMinMax`

- *Nearest neighbors*
- *Hyper-parameter tuning of a Pipeline with KNeighborsTimeSeriesClassifier*
- *Soft-DTW weighted barycenters*
- *SVM and GAK*
- *Learning Shapelets*
- *Aligning discovered shapelets with timeseries*
- *Learning Shapelets: decision boundaries in 2D distance space*

3.11.3 `tslearn.preprocessing`.TimeSeriesResampler

class tslearn.preprocessing.**TimeSeriesResampler**(*sz*)

Resampler for time series. Resample time series so that they reach the target size.

Parameters

sz [int] Size of the output time series.

Examples

```
>>> TimeSeriesResampler(sz=5).fit_transform([[0, 3, 6]])
array([[[0. ],
        [1.5],
        [3. ],
        [4.5],
        [6. ]]])
```

Methods

fit(X[, y])	A dummy method such that it complies to the sklearn requirements.
fit_transform(X[, y])	Fit to data, then transform it.
transform(X[, y])	Fit to data, then transform it.

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

A dummy method such that it complies to the sklearn requirements. Since this method is completely stateless, it just returns itself.

Parameters

X Ignored

Returns

self

fit_transform(*X*, *y=None*, ***kwargs*)
Fit to data, then transform it.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset to be resampled.

Returns

numpy.ndarray Resampled time series dataset.

transform(*X*, *y=None*, ***kwargs*)
Fit to data, then transform it.

Parameters

X [array-like of shape (n_ts, sz, d)] Time series dataset to be resampled.

Returns

numpy.ndarray Resampled time series dataset.

Examples using `tslearn.preprocessing.TimeSeriesResampler`

- *k-means*

3.12 tslearn.shapelets

The *tslearn.shapelets* module gathers Shapelet-based algorithms.

It depends on the *tensorflow* library for optimization (TF2 is required).

User guide: See the *Shapelets* section for further details.

Functions

grabocka_params_to_shapelet_size_dict(n_ts, ts_sz, ...)	Compute number and length of shapelets.

3.12.1 tslearn.shapelets.grabocka_params_to_shapelet_size_dict

tslearn.shapelets.**grabocka_params_to_shapelet_size_dict**(*n_ts*, *ts_sz*, *n_classes*, *l*, *r*)
Compute number and length of shapelets.

This function uses the heuristic from [1].

Parameters

n_ts: int Number of time series in the dataset

ts_sz: int Length of time series in the dataset

n_classes: int Number of classes in the dataset

l: float Fraction of the length of time series to be used for base shapelet length

r: int Number of different shapelet lengths to use

Returns

dict Dictionary giving, for each shapelet length, the number of such shapelets to be generated

References

[1]

Examples

```
>>> d = grabocka_params_to_shapelet_size_dict(
...         n_ts=100, ts_sz=100, n_classes=3, l=0.1, r=2)
>>> keys = sorted(d.keys())
>>> print(keys)
[10, 20]
>>> print([d[k] for k in keys])
[4, 4]
```

Examples using `tslearn.shapelets.grabocka_params_to_shapelet_size_dict`

- *Learning Shapelets*

Classes

LearningShapelets([n_shapelets_per_size, ...])	Learning Time-Series Shapelets model.

3.12.2 `tslearn.shapelets.LearningShapelets`

class tslearn.shapelets.**LearningShapelets**(*n_shapelets_per_size=None*, *max_iter=10000*, *batch_size=256*, *verbose=0*, *optimizer='sgd'*, *weight_regularizer=0.0*, *shapelet_length=0.15*, *total_lengths=3*, *max_size=None*, *scale=False*, *random_state=None*)

Learning Time-Series Shapelets model.

Learning Time-Series Shapelets was originally presented in [1].

From an input (possibly multidimensional) time series x and a set of shapelets $\{s_i\}_i$, the i-th coordinate of the Shapelet transform is computed as:

$$ST(x, s_i) = \min_t \sum_{\delta_t} \|x(t + \delta_t) - s_i(\delta_t)\|_2^2$$

The Shapelet model consists in a logistic regression layer on top of this transform. Shapelet coefficients as well as logistic regression weights are optimized by gradient descent on a L2-penalized cross-entropy loss.

Parameters

n_shapelets_per_size: dict (default: None) Dictionary giving, for each shapelet size (key), the number of such shapelets to be trained (value). If None, *grabocka_params_to_shapelet_size_dict* is used and the size used to compute is that of the shortest time series passed at fit time.

max_iter: int (default: 10,000) Number of training epochs.

Changed in version 0.3: default value for max_iter is set to 10,000 instead of 100

batch_size: int (default: 256) Batch size to be used.

verbose: {0, 1, 2} (default: 0) *keras* verbose level.

optimizer: str or keras.optimizers.Optimizer (default: "sgd") *keras* optimizer to use for training.

weight_regularizer: float or None (default: 0.) Strength of the L2 regularizer to use for training the classification (softmax) layer. If 0, no regularization is performed.

shapelet_length: float (default: 0.15) The length of the shapelets, expressed as a fraction of the time series length. Used only if *n_shapelets_per_size* is None.

total_lengths: int (default: 3) The number of different shapelet lengths. Will extract shapelets of length i * shapelet_length for i in [1, total_lengths] Used only if *n_shapelets_per_size* is None.

max_size: int or None (default: None) Maximum size for time series to be fed to the model. If None, it is set to the size (number of timestamps) of the training time series.

scale: bool (default: False) Whether input data should be scaled for each feature of each time series to lie in the [0-1] interval. Default for this parameter is set to *False* in version 0.4 to ensure backward compatibility, but is likely to change in a future version.

random_state [int or None, optional (default: None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Attributes

shapelets_ [numpy.ndarray of objects, each object being a time series] Set of time-series shapelets.

shapelets_as_time_series_ [numpy.ndarray of shape (n_shapelets, sz_shp, d) where *sz_shp* is the maximum of all shapelet sizes] Set of time-series shapelets formatted as a `tslearn` time series dataset.

transformer_model_ [keras.Model] Transforms an input dataset of timeseries into distances to the learned shapelets.

locator_model_ [keras.Model] Returns the indices where each of the shapelets can be found (minimal distance) within each of the timeseries of the input dataset.

model_ [keras.Model] Directly predicts the class probabilities for the input timeseries.

history_ [dict] Dictionary of losses and metrics recorded during fit.

References

[1]

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=16, d=2, n_blobs=3)
>>> clf = LearningShapelets(n_shapelets_per_size={4: 5},
...                         max_iter=1, verbose=0)
>>> clf.fit(X, y).shapelets_.shape
(5,)
>>> clf.shapelets_[0].shape
(4, 2)
>>> clf.predict(X).shape
(30,)
>>> clf.predict_proba(X).shape
(30, 3)
>>> clf.transform(X).shape
(30, 5)
```

Methods

fit(X, y)	Learn time-series shapelets.
fit_transform(X[, y])	Fit to data, then transform it.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_params([deep])	Get parameters for this estimator.
get_weights([layer_name])	Return model weights (or weights for a given layer if *layer_name* is provided).
locate(X)	Compute shapelet match location for a set of time series.
predict(X)	Predict class for a given set of time series.
predict_proba(X)	Predict class probability for a given set of time series.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
set_weights(weights[, layer_name])	Set model weights (or weights for a given layer if *layer_name* is provided).
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X)	Generate shapelet transform for a set of time series.

fit (*X*, *y*)
Learn time-series shapelets.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

y [array-like of shape=(n_ts,)] Time series labels.

fit_transform (*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [{array-like, sparse matrix, dataframe} of shape (n_samples, n_features)]

y [ndarray of shape (n_samples,), default=None] Target values.

****fit_params** [dict] Additional fit parameters.

Returns

X_new [ndarray array of shape (n_samples, n_features_new)] Transformed array.

classmethod from_hdf5(*path*)
Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)
Load model from a JSON file.

Parameters

path [str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)
Load model from a pickle file.

Parameters

path [str] Full path to file.

Returns

Model instance

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_weights(*layer_name=None*)
Return model weights (or weights for a given layer if *layer_name* is provided).

Parameters

layer_name: str or None (default: None) Name of the layer for which weights should be returned. If None, all model weights are returned. Available layer names with weights are:

- "shapelets_i_j" with i an integer for the shapelet id and j an integer for the dimension
- "classification" for the final classification layer

Returns

list list of model (or layer) weights

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=100, sz=256, d=1, n_blobs=3)
>>> clf = LearningShapelets(n_shapelets_per_size={10: 5}, max_iter=0,
...                         verbose=0)
>>> clf.fit(X, y).get_weights("classification")[0].shape
(5, 3)
>>> clf.get_weights("shapelets_0_0")[0].shape
(5, 10)
>>> len(clf.get_weights("shapelets_0_0"))
1
```

locate(*X*)

Compute shapelet match location for a set of time series.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_shapelets) Location of the shapelet matches for the provided time series.

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X = numpy.zeros((3, 10, 1))
>>> X[0, 4:7, 0] = numpy.array([1, 2, 3])
>>> y = [1, 0, 0]
>>> # Data is all zeros except a motif 1-2-3 in the first time series
>>> clf = LearningShapelets(n_shapelets_per_size={3: 1}, max_iter=0,
...                         verbose=0)
>>> _ = clf.fit(X, y)
>>> weights_shapelet = [
...     numpy.array([[1, 2, 3]])
... ]
>>> clf.set_weights(weights_shapelet, layer_name="shapelets_0_0")
>>> clf.locate(X)
array([[4],
       [0],
       [0]])
```

predict(*X*)

Predict class for a given set of time series.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts,) or (n_ts, n_classes), depending on the shape

of the label vector provided at training time. Index of the cluster each sample belongs to or class probability matrix, depending on what was provided at training time.

predict_proba(*X*)
Predict class probability for a given set of time series.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_classes), Class probability matrix.

score(*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

set_weights(*weights*, *layer_name=None*)
Set model weights (or weights for a given layer if *layer_name* is provided).

Parameters

weights: list of ndarrays Weights to set for the model / target layer

layer_name: str or None (default: None) Name of the layer for which weights should be set. If None, all model weights are set. Available layer names with weights are:

- "shapelets_i_j" with i an integer for the shapelet id and j an integer for the dimension
- "classification" for the final classification layer

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=16, d=1, n_blobs=3)
>>> clf = LearningShapelets(n_shapelets_per_size={3: 1}, max_iter=0,
...                         verbose=0)
```

(continues on next page)

(continued from previous page)

```
>>> _ = clf.fit(X, y)
>>> weights_shapelet = [
...     numpy.array([[1, 2, 3]])
... ]
>>> clf.set_weights(weights_shapelet, layer_name="shapelets_0_0")
>>> clf.shapelets_as_time_series_
array([[[1.],
        [2.],
        [3.]]])
```

shapelets_as_time_series_
Set of time-series shapelets formatted as a `tslearn` time series dataset.

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=256, d=1, n_blobs=3)
>>> model = LearningShapelets(n_shapelets_per_size={3: 2, 4: 1},
...                           max_iter=1)
>>> _ = model.fit(X, y)
>>> model.shapelets_as_time_series_.shape
(3, 4, 1)
```

to_hdf5(*path*)
Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path [str] Full file path. File must not already exist.

Raises

FileExistsError If a file with the same path already exists.

to_json(*path*)
Save model to a JSON file.

Parameters

path [str] Full file path.

to_pickle(*path*)
Save model to a pickle file.

Parameters

path [str] Full file path.

transform(*X*)
Generate shapelet transform for a set of time series.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_shapelets) Shapelet-Transform of the provided time series.

Examples using `tslearn.shapelets.LearningShapelets`

- *Learning Shapelets*
- *Aligning discovered shapelets with timeseries*
- *Learning Shapelets: decision boundaries in 2D distance space*

3.13 tslearn.svm

The `tslearn.svm` module contains Support Vector Classifier (SVC) and Support Vector Regressor (SVR) models for time series.

Classes

`TimeSeriesSVC`([C, kernel, degree, gamma, ...])	Time-series specific Support Vector Classifier.
`TimeSeriesSVR`([C, kernel, degree, gamma, ...])	Time-series specific Support Vector Regressor.

3.13.1 `tslearn.svm.TimeSeriesSVC`

class tslearn.svm.**TimeSeriesSVC**(*C=1.0*, *kernel='gak'*, *degree=3*, *gamma='auto'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *n_jobs=None*, *verbose=0*, *max_iter=-1*, *decision_function_shape='ovr'*, *random_state=None*)

Time-series specific Support Vector Classifier.

Parameters

C [float, optional (default=1.0)] Penalty parameter C of the error term.

kernel [string, optional (default='gak')] Specifies the kernel type to be used in the algorithm. It must be one of 'gak' or a kernel accepted by `sklearn.svm.SVC`. If none is given, 'gak' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [float, optional (default='auto')] Kernel coefficient for 'gak', 'rbf', 'poly' and 'sigmoid'. If gamma is 'auto' then:

- for 'gak' kernel, it is computed based on a sampling of the training set (cf *tslearn.metrics.gamma_soft_dtw*)
- for other kernels (eg. 'rbf'), 1/n_features will be used.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

probability [boolean, optional (default=False)] Whether to enable probability estimates. This must be enabled prior to calling *fit*, and will slow down that method. Also, probability estimates are not guaranteed to match predict output. See our *dedicated user guide section* for more details.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

cache_size [float, optional (default=200.0)] Specify the size of the kernel cache (in MB).

class_weight [{dict, 'balanced'}, optional] Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for GAK cross-similarity matrix computations. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose [int, default: 0] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape ['ovo', 'ovr', default='ovr'] Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2).

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Attributes

support_ [array-like, shape = [n_SV]] Indices of support vectors.

n_support_ [array-like, dtype=int32, shape = [n_class]] Number of support vectors for each class.

support_vectors_ [list of arrays of shape [n_SV, sz, d]] List of support vectors in tslearn dataset format, one array per class

dual_coef_ [array, shape = [n_class-1, n_SV]] Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide of `sklearn` for details.

coef_ [array, shape = [n_class-1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel. *coef_* is a readonly property derived from *dual_coef_* and *support_vectors_*.

intercept_ [array, shape = [n_class * (n_class-1) / 2]] Constants in decision function.

svm_estimator_ [sklearn.svm.SVC] The underlying sklearn estimator

References

Fast Global Alignment Kernels. Marco Cuturi. ICML 2011.

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=64, d=2, n_blobs=2)
>>> clf = TimeSeriesSVC(kernel="gak", gamma="auto", probability=True)
>>> clf.fit(X, y).predict(X).shape
(20,)
>>> sv = clf.support_vectors_
>>> len(sv)  # should be equal to the nr of classes in the clf problem
2
>>> sv[0].shape  # doctest: +ELLIPSIS
(..., 64, 2)
>>> sv_sum = sum([sv_i.shape[0] for sv_i in sv])
>>> sv_sum == clf.svm_estimator_.n_support_.sum()
True
>>> clf.decision_function(X).shape
(20,)
>>> clf.predict_log_proba(X).shape
(20, 2)
>>> clf.predict_proba(X).shape
(20, 2)
```

Methods

decision_function(X)	Evaluates the decision function for the samples in X.
fit(X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class for a given set of time series.
predict_log_proba(X)	Predict class log-probabilities for a given set of time series.
predict_proba(X)	Predict class probability for a given set of time series.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
*support_vectors_time_series_*([X])	DEPRECATED: The use of *support_vectors_time_series_* is deprecated in tslearn v0.4 and will be removed in v0.6.

decision_function(*X*)
Evaluates the decision function for the samples in X.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

ndarray of shape (n_samples, n_classes * (n_classes-1) / 2) Returns the decision function of the sample for each class in the model. If decision_function_shape='ovr', the shape is (n_samples, n_classes).

fit(*X*, *y*, *sample_weight=None*)
Fit the SVM model according to the given training data.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

y [array-like of shape=(n_ts,)] Time series labels.

sample_weight [array-like of shape (n_samples,), default=None] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*X*)
Predict class for a given set of time series.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts,) or (n_ts, n_classes), depending on the shape

of the label vector provided at training time. Index of the cluster each sample belongs to or class probability matrix, depending on what was provided at training time.

predict_log_proba(*X*)
Predict class log-probabilities for a given set of time series.

Note that probability estimates are not guaranteed to match predict output. See our *dedicated user guide section* for more details.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_classes), Class probability matrix.

predict_proba(*X*)
Predict class probability for a given set of time series.

Note that probability estimates are not guaranteed to match predict output. See our *dedicated user guide section* for more details.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_classes), Class probability matrix.

score(*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

support_vectors_time_series_(*X=None*)
DEPRECATED: The use of *support_vectors_time_series_* is deprecated in tslearn v0.4 and will be removed in v0.6. Use *support_vectors_* property instead.

Examples using `tslearn.svm.TimeSeriesSVC`

- *SVM and GAK*

3.13.2 tslearn.svm.TimeSeriesSVR

class tslearn.svm.**TimeSeriesSVR**(*C=1.0*, *kernel='gak'*, *degree=3*, *gamma='auto'*, *coef0=0.0*, *tol=0.001*, *epsilon=0.1*, *shrinking=True*, *cache_size=200*, *n_jobs=None*, *verbose=0*, *max_iter=-1*)
Time-series specific Support Vector Regressor.

Parameters

C [float, optional (default=1.0)] Penalty parameter C of the error term.

kernel [string, optional (default='gak')] Specifies the kernel type to be used in the algorithm. It must be one of 'gak' or a kernel accepted by `sklearn.svm.SVC`. If none is given, 'gak' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [float, optional (default='auto')] Kernel coefficient for 'gak', 'rbf', 'poly' and 'sigmoid'. If gamma is 'auto' then:

- for 'gak' kernel, it is computed based on a sampling of the training set (cf *tslearn.metrics.gamma_soft_dtw*)
- for other kernels (eg. 'rbf'), 1/n_features will be used.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

epsilon [float, optional (default=0.1)] Epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

cache_size [float, optional (default=200.0)] Specify the size of the kernel cache (in MB).

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for GAK cross-similarity matrix computations. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose [int, default: 0] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

Attributes

support_ [array-like, shape = [n_SV]] Indices of support vectors.

support_vectors_ [array of shape [n_SV, sz, d]] Support vectors in tslearn dataset format

dual_coef_ [array, shape = [1, n_SV]] Coefficients of the support vector in the decision function.

coef_ [array, shape = [1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel. *coef_* is readonly property derived from *dual_coef_* and *support_vectors_*.

intercept_ [array, shape = [1]] Constants in decision function.

sample_weight [array-like, shape = [n_samples]] Individual weights for each sample

svm_estimator_ [sklearn.svm.SVR] The underlying sklearn estimator

References

Fast Global Alignment Kernels. Marco Cuturi. ICML 2011.

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=64, d=2, n_blobs=2)
>>> import numpy
>>> y = y.astype(numpy.float) + numpy.random.randn(20) * .1
>>> reg = TimeSeriesSVR(kernel="gak", gamma="auto")
>>> reg.fit(X, y).predict(X).shape
(20,)
>>> sv = reg.support_vectors_
>>> sv.shape  # doctest: +ELLIPSIS
(..., 64, 2)
```

(continues on next page)

(continued from previous page)

```
>>> sv.shape[0] <= 20
True
```

Methods

fit(X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class for a given set of time series.
score(X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.
*support_vectors_time_series_*([X])	DEPRECATED: The use of *support_vectors_time_series_* is deprecated in tslearn v0.4 and will be removed in v0.6.

fit (*X*, *y*, *sample_weight=None*)
Fit the SVM model according to the given training data.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

y [array-like of shape=(n_ts,)] Time series labels.

sample_weight [array-like of shape (n_samples,), default=None] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

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

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*X*)
Predict class for a given set of time series.

Parameters

X [array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts,) or (n_ts, dim_output), depending on the shape

of the target vector provided at training time. Predicted targets

score (*X*, *y*, *sample_weight=None*)
Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score

is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

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

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [object] Estimator instance.

support_vectors_time_series_(*X=None*)
DEPRECATED: The use of *support_vectors_time_series_* is deprecated in tslearn v0.4 and will be removed in v0.6. Use *support_vectors_* property instead.

3.14 tslearn.utils

The *tslearn.utils* module includes various utilities.

Generic functions

to_time_series(ts[, remove_nans])	Transforms a time series so that it fits the format used in `tslearn` models.
to_time_series_dataset(dataset[, dtype])	Transforms a time series dataset so that it fits the format used in `tslearn` models.
to_sklearn_dataset(dataset[, dtype, return_dim])	Transforms a time series dataset so that it fits the format used in `sklearn` estimators.

Continued on next page

Table 38 – continued from previous page

ts_size(ts)	Returns actual time series size.
ts_zeros(sz[, d])	Returns a time series made of zero values.
load_time_series_txt(fname)	Loads a time series dataset from disk.
save_time_series_txt(fname, dataset[, fmt])	Writes a time series dataset to disk.
check_equal_size(dataset)	Check if all time series in the dataset have the same size.
check_dims(X[, X_fit_dims, extend, ...])	Reshapes X to a 3-dimensional array of X.shape[0] univariate timeseries of length X.shape[1] if X is 2-dimensional and extend is True.

3.14.1 tslearn.utils.to_time_series

tslearn.utils.**to_time_series**(*ts*, *remove_nans=False*)

Transforms a time series so that it fits the format used in tslearn models.

Parameters

ts [array-like] The time series to be transformed.

remove_nans [bool (default: False)] Whether trailing NaNs at the end of the time series should be removed or not

Returns

numpy.ndarray of shape (sz, d) The transformed time series.

See also:

to_time_series_dataset Transforms a dataset of time series

Examples

```
>>> to_time_series([1, 2])
array([[1.],
       [2.]])
>>> to_time_series([1, 2, numpy.nan])
array([[ 1.],
       [ 2.],
       [nan]])
>>> to_time_series([1, 2, numpy.nan], remove_nans=True)
array([[1.],
       [2.]])
```

3.14.2 tslearn.utils.to_time_series_dataset

tslearn.utils.**to_time_series_dataset**(*dataset*, *dtype=<class 'float'>*)

Transforms a time series dataset so that it fits the format used in tslearn models.

Parameters

dataset [array-like] The dataset of time series to be transformed.

dtype [data type (default: numpy.float)] Data type for the returned dataset.

Returns

numpy.ndarray of shape (n_ts, sz, d) The transformed dataset of time series.

See also:

to_time_series Transforms a single time series

Examples

```
>>> to_time_series_dataset([[1, 2]])
array([[[1.],
        [2.]]])
>>> to_time_series_dataset([[1, 2], [1, 4, 3]])
array([[[ 1.],
        [ 2.],
        [nan]],
<BLANKLINE>
       [[ 1.],
        [ 4.],
        [ 3.]]])
```

3.14.3 tslearn.utils.to_sklearn_dataset

tslearn.utils.**to_sklearn_dataset**(*dataset*, *dtype=<class 'float'>*, *return_dim=False*)

Transforms a time series dataset so that it fits the format used in sklearn estimators.

Parameters

dataset [array-like] The dataset of time series to be transformed.

dtype [data type (default: numpy.float)] Data type for the returned dataset.

return_dim [boolean (optional, default: False)] Whether the dimensionality (third dimension should be returned together with the transformed dataset).

Returns

numpy.ndarray of shape (n_ts, sz * d) The transformed dataset of time series.

int (optional, if return_dim=True) The dimensionality of the original tslearn dataset (third dimension)

See also:

to_time_series_dataset Transforms a time series dataset to tslearn **format.**

Examples

```
>>> to_sklearn_dataset([[1, 2]], return_dim=True)
(array([[1., 2.]]), 1)
>>> to_sklearn_dataset([[1, 2], [1, 4, 3]])
array([[ 1.,  2., nan],
       [ 1.,  4.,  3.]])
```

3.14.4 tslearn.utils.ts_size

tslearn.utils.**ts_size**(*ts*)
Returns actual time series size.

Final timesteps that have NaN values for all dimensions will be removed from the count.

Parameters

ts [array-like] A time series.

Returns

int Actual size of the time series.

Examples

```
>>> ts_size([1, 2, 3, numpy.nan])
3
>>> ts_size([1, numpy.nan])
1
>>> ts_size([numpy.nan])
0
>>> ts_size([[1, 2],
...          [2, 3],
...          [3, 4],
...          [numpy.nan, 2],
...          [numpy.nan, numpy.nan]])
4
```

Examples using `tslearn.utils.ts_size`

- *Learning Shapelets*

3.14.5 tslearn.utils.ts_zeros

tslearn.utils.**ts_zeros**(*sz*, *d=1*)
Returns a time series made of zero values.

Parameters

sz [int] Time series size.

d [int (optional, default: 1)] Time series dimensionality.

Returns

numpy.ndarray A time series made of zeros.

Examples

```
>>> ts_zeros(3, 2)  # doctest: +NORMALIZE_WHITESPACE
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

(continues on next page)

(continued from previous page)

```
>>> ts_zeros(5).shape
(5, 1)
```

3.14.6 tslearn.utils.load_time_series_txt

tslearn.utils.**load_time_series_txt**(*fname*)

Loads a time series dataset from disk.

Parameters

fname [string] Path to the file from which time series should be read.

Returns

numpy.ndarray or array of numpy.ndarray The dataset of time series.

See also:

save_time_series_txt Save time series to disk

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4], [1, 2, 3]])
>>> save_time_series_txt("tmp-tslearn-test.txt", dataset)
>>> reloaded_dataset = load_time_series_txt("tmp-tslearn-test.txt")
```

3.14.7 tslearn.utils.save_time_series_txt

tslearn.utils.**save_time_series_txt**(*fname*, *dataset*, *fmt='%.18e'*)

Writes a time series dataset to disk.

Parameters

fname [string] Path to the file in which time series should be written.

dataset [array-like] The dataset of time series to be saved.

fmt [string (default: "%.18e")] Format to be used to write each value.

See also:

load_time_series_txt Load time series from disk

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4], [1, 2, 3]])
>>> save_time_series_txt("tmp-tslearn-test.txt", dataset)
```

3.14.8 tslearn.utils.check_equal_size

tslearn.utils.**check_equal_size**(*dataset*)

Check if all time series in the dataset have the same size.

Parameters

dataset: array-like The dataset to check.

Returns

bool Whether all time series in the dataset have the same size.

Examples

```
>>> check_equal_size([[1, 2, 3], [4, 5, 6], [5, 3, 2]])
True
>>> check_equal_size([[1, 2, 3, 4], [4, 5, 6], [5, 3, 2]])
False
```

3.14.9 tslearn.utils.check_dims

tslearn.utils.**check_dims**(*X*, *X_fit_dims=None*, *extend=True*, *check_n_features_only=False*)

Reshapes X to a 3-dimensional array of X.shape[0] univariate timeseries of length X.shape[1] if X is 2-dimensional and extend is True. Then checks whether the provided X_fit_dims and the dimensions of X (except for the first one), match.

Parameters

X [array-like] The first array to be compared.

X_fit_dims [tuple (default: None)] The dimensions of the data generated by fit, to compare with the dimensions of the provided array X. If None, then only perform reshaping of X, if necessary.

extend [boolean (default: True)] Whether to reshape X, if it is 2-dimensional.

check_n_features_only: boolean (default: False)

Returns

array Reshaped X array

Raises

ValueError Will raise exception if X is None or (if X_fit_dims is provided) one of the dimensions of the provided data, except the first, does not match X_fit_dims.

Examples

```
>>> X = numpy.empty((10, 3))
>>> check_dims(X).shape
(10, 3, 1)
>>> X = numpy.empty((10, 3, 1))
>>> check_dims(X).shape
(10, 3, 1)
>>> X_fit_dims = (5, 3, 1)
```

(continues on next page)

(continued from previous page)

```
>>> check_dims(X, X_fit_dims).shape
(10, 3, 1)
>>> X_fit_dims = (5, 3, 2)
>>> check_dims(X, X_fit_dims)  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
ValueError: Dimensions (except first) must match! ((5, 3, 2) and (10, 3, 1)
are passed shapes)
>>> X_fit_dims = (5, 5, 1)
>>> check_dims(X, X_fit_dims, check_n_features_only=True).shape
(10, 3, 1)
>>> X_fit_dims = (5, 5, 2)
>>> check_dims(
...     X,
...     X_fit_dims,
...     check_n_features_only=True
... )  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
ValueError: Number of features of the provided timeseries must match!
(last dimension) must match the one of the fitted data!
((5, 5, 2) and (10, 3, 1) are passed shapes)
```

Conversion functions

The following functions are provided for the sake of interoperability between standard Python packages for time series. They allow conversion between *tslearn* format and other libraries' formats.

to_pyts_dataset(X)	Transform a tslearn-compatible dataset into a pyts dataset.
from_pyts_dataset(X)	Transform a pyts-compatible dataset into a tslearn dataset.
to_sktime_dataset(X)	Transform a tslearn-compatible dataset into a sktime dataset.
from_sktime_dataset(X)	Transform a sktime-compatible dataset into a tslearn dataset.
to_cesium_dataset(X)	Transform a tslearn-compatible dataset into a cesium dataset.
from_cesium_dataset(X)	Transform a cesium-compatible dataset into a tslearn dataset.
to_seglearn_dataset(X)	Transform a tslearn-compatible dataset into a seglearn dataset.
from_seglearn_dataset(X)	Transform a seglearn-compatible dataset into a tslearn dataset.
to_tsfresh_dataset(X)	Transform a tslearn-compatible dataset into a tsfresh dataset.
from_tsfresh_dataset(X)	Transform a tsfresh-compatible dataset into a tslearn dataset.
to_stumpy_dataset(X)	Transform a tslearn-compatible dataset into a stumpy dataset.
from_stumpy_dataset(X)	Transform a stumpy-compatible dataset into a tslearn dataset.

Continued on next page

Table 39 – continued from previous page

to_pyflux_dataset(X)	Transform a tslearn-compatible dataset into a pyflux dataset.
from_pyflux_dataset(X)	Transform a pyflux-compatible dataset into a tslearn dataset.

3.14.10 tslearn.utils.to_pyts_dataset

tslearn.utils.**to_pyts_dataset**(*X*)

Transform a tslearn-compatible dataset into a pyts dataset.

Parameters

X: array, shape = (n_ts, sz, d) tslearn-formatted dataset to be cast to pyts format

Returns

array, shape=(n_ts, sz) if d=1, (n_ts, d, sz) otherwise pyts-formatted dataset

Examples

```
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> pyts_arr = to_pyts_dataset(tslearn_arr)
>>> pyts_arr.shape
(10, 16)
>>> tslearn_arr = numpy.random.randn(10, 16, 2)
>>> pyts_arr = to_pyts_dataset(tslearn_arr)
>>> pyts_arr.shape
(10, 2, 16)
>>> tslearn_arr = [numpy.random.randn(16, 1), numpy.random.randn(10, 1)]
>>> to_pyts_dataset(tslearn_arr)  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
...
ValueError: All the time series in the array should be of equal lengths
```

3.14.11 tslearn.utils.from_pyts_dataset

tslearn.utils.**from_pyts_dataset**(*X*)

Transform a pyts-compatible dataset into a tslearn dataset.

Parameters

X: array, shape = (n_ts, sz) or (n_ts, d, sz) pyts-formatted dataset

Returns

array, shape=(n_ts, sz, d) tslearn-formatted dataset

Examples

```
>>> pyts_arr = numpy.random.randn(10, 16)
>>> tslearn_arr = from_pyts_dataset(pyts_arr)
>>> tslearn_arr.shape
(10, 16, 1)
```

(continues on next page)

(continued from previous page)

```
>>> pyts_arr = numpy.random.randn(10, 2, 16)
>>> tslearn_arr = from_pyts_dataset(pyts_arr)
>>> tslearn_arr.shape
(10, 16, 2)
>>> pyts_arr = numpy.random.randn(10)
>>> from_pyts_dataset(pyts_arr)  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
...
ValueError: X is not a valid input pyts array.
```

3.14.12 tslearn.utils.to_sktime_dataset

tslearn.utils.**to_sktime_dataset**(*X*)

Transform a tslearn-compatible dataset into a sktime dataset.

Parameters

X: array, shape = (n_ts, sz, d) tslearn-formatted dataset to be cast to sktime format

Returns

Pandas data-frame sktime-formatted dataset (cf. link)

Notes

Conversion from/to sktime format requires pandas to be installed.

Examples

```
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> sktime_arr = to_sktime_dataset(tslearn_arr)
>>> sktime_arr.shape
(10, 1)
>>> sktime_arr["dim_0"][0].shape
(16,)
>>> tslearn_arr = numpy.random.randn(10, 16, 2)
>>> sktime_arr = to_sktime_dataset(tslearn_arr)
>>> sktime_arr.shape
(10, 2)
>>> sktime_arr["dim_1"][0].shape
(16,)
```

3.14.13 tslearn.utils.from_sktime_dataset

tslearn.utils.**from_sktime_dataset**(*X*)

Transform a sktime-compatible dataset into a tslearn dataset.

Parameters

X: pandas data-frame sktime-formatted dataset (cf. link)

Returns

array, shape=(n_ts, sz, d) tslearn-formatted dataset

Notes

Conversion from/to sktime format requires pandas to be installed.

Examples

```
>>> import pandas as pd
>>> sktime_df = pd.DataFrame()
>>> sktime_df["dim_0"] = [pd.Series([1, 2, 3]), pd.Series([4, 5, 6])]
>>> tslearn_arr = from_sktime_dataset(sktime_df)
>>> tslearn_arr.shape
(2, 3, 1)
>>> sktime_df = pd.DataFrame()
>>> sktime_df["dim_0"] = [pd.Series([1, 2, 3]),
...                       pd.Series([4, 5, 6, 7])]
>>> sktime_df["dim_1"] = [pd.Series([8, 9, 10]),
...                       pd.Series([11, 12, 13, 14])]
>>> tslearn_arr = from_sktime_dataset(sktime_df)
>>> tslearn_arr.shape
(2, 4, 2)
>>> sktime_arr = numpy.random.randn(10, 1, 16)
>>> from_sktime_dataset(
...     sktime_arr
... )  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
...
ValueError: X is not a valid input sktime array.
```

3.14.14 tslearn.utils.to_cesium_dataset

`tslearn.utils.`**`to_cesium_dataset`**`(X)`

Transform a tslearn-compatible dataset into a cesium dataset.

Parameters

X: array, shape = (n_ts, sz, d), where n_ts=1 tslearn-formatted dataset to be cast to cesium format

Returns

list of cesium TimeSeries cesium-formatted dataset (cf. link)

Notes

Conversion from/to cesium format requires cesium to be installed.

Examples

```
>>> tslearn_arr = numpy.random.randn(3, 16, 1)
>>> cesium_ds = to_cesium_dataset(tslearn_arr)
>>> len(cesium_ds)
3
>>> cesium_ds[0].measurement.shape
```

(continues on next page)

(continued from previous page)

```
(16,)
>>> tslearn_arr = numpy.random.randn(3, 16, 2)
>>> cesium_ds = to_cesium_dataset(tslearn_arr)
>>> len(cesium_ds)
3
>>> cesium_ds[0].measurement.shape
(2, 16)
>>> tslearn_arr = [[1, 2, 3], [1, 2, 3, 4]]
>>> cesium_ds = to_cesium_dataset(tslearn_arr)
>>> len(cesium_ds)
2
>>> cesium_ds[0].measurement.shape
(3,)
```

3.14.15 tslearn.utils.from_cesium_dataset

tslearn.utils.**from_cesium_dataset**(*X*)

Transform a cesium-compatible dataset into a tslearn dataset.

Parameters

X: list of cesium TimeSeries cesium-formatted dataset (cf. link)

Returns

array, shape=(n_ts, sz, d) tslearn-formatted dataset.

Notes

Conversion from/to cesium format requires cesium to be installed.

Examples

```
>>> from cesium.time_series import TimeSeries
>>> cesium_ds = [TimeSeries(m=numpy.array([1, 2, 3, 4]))]
>>> tslearn_arr = from_cesium_dataset(cesium_ds)
>>> tslearn_arr.shape
(1, 4, 1)
>>> cesium_ds = [
...     TimeSeries(m=numpy.array([[1, 2, 3, 4],
...                               [5, 6, 7, 8]]))
... ]
>>> tslearn_arr = from_cesium_dataset(cesium_ds)
>>> tslearn_arr.shape
(1, 4, 2)
```

3.14.16 tslearn.utils.to_seglearn_dataset

tslearn.utils.**to_seglearn_dataset**(*X*)

Transform a tslearn-compatible dataset into a seglearn dataset.

Parameters

X: array, shape = (n_ts, sz, d) tslearn-formatted dataset to be cast to seglearn format

Returns

array of arrays, shape=(n_ts,) seglearn-formatted dataset. i-th sub-array in the list has shape (sz_i, d)

Examples

```
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> seglearn_arr = to_seglearn_dataset(tslearn_arr)
>>> seglearn_arr.shape
(10, 16, 1)
>>> tslearn_arr = numpy.random.randn(10, 16, 2)
>>> seglearn_arr = to_seglearn_dataset(tslearn_arr)
>>> seglearn_arr.shape
(10, 16, 2)
>>> tslearn_arr = [numpy.random.randn(16, 2), numpy.random.randn(10, 2)]
>>> seglearn_arr = to_seglearn_dataset(tslearn_arr)
>>> seglearn_arr.shape
(2,)
>>> seglearn_arr[0].shape
(16, 2)
>>> seglearn_arr[1].shape
(10, 2)
```

3.14.17 tslearn.utils.from_seglearn_dataset

tslearn.utils.**from_seglearn_dataset**(*X*)

Transform a seglearn-compatible dataset into a tslearn dataset.

Parameters

X: list of arrays, or array of arrays, shape = (n_ts,) seglearn-formatted dataset. i-th sub-array in the list has shape (sz_i, d)

Returns

array, shape=(n_ts, sz, d), where sz is the maximum of all array lengths tslearn-formatted dataset

Examples

```
>>> seglearn_arr = [numpy.random.randn(10, 1), numpy.random.randn(10, 1)]
>>> tslearn_arr = from_seglearn_dataset(seglearn_arr)
>>> tslearn_arr.shape
(2, 10, 1)
>>> seglearn_arr = [numpy.random.randn(10, 1), numpy.random.randn(5, 1)]
>>> tslearn_arr = from_seglearn_dataset(seglearn_arr)
>>> tslearn_arr.shape
(2, 10, 1)
>>> seglearn_arr = numpy.random.randn(2, 10, 1)
>>> tslearn_arr = from_seglearn_dataset(seglearn_arr)
>>> tslearn_arr.shape
(2, 10, 1)
```

3.14.18 tslearn.utils.to_tsfresh_dataset

tslearn.utils.**to_tsfresh_dataset**(*X*)

Transform a tslearn-compatible dataset into a tsfresh dataset.

Parameters

X: array, shape = (n_ts, sz, d) tslearn-formatted dataset to be cast to tsfresh format

Returns

Pandas data-frame tsfresh-formatted dataset ("flat" data frame, as described there)

Notes

Conversion from/to tsfresh format requires pandas to be installed.

Examples

```
>>> tslearn_arr = numpy.random.randn(1, 16, 1)
>>> tsfresh_df = to_tsfresh_dataset(tslearn_arr)
>>> tsfresh_df.shape
(16, 3)
>>> tslearn_arr = numpy.random.randn(1, 16, 2)
>>> tsfresh_df = to_tsfresh_dataset(tslearn_arr)
>>> tsfresh_df.shape
(16, 4)
```

3.14.19 tslearn.utils.from_tsfresh_dataset

tslearn.utils.**from_tsfresh_dataset**(*X*)

Transform a tsfresh-compatible dataset into a tslearn dataset.

Parameters

X: pandas data-frame tsfresh-formatted dataset ("flat" data frame, as described there)

Returns

array, shape=(n_ts, sz, d) tslearn-formatted dataset. Column order is kept the same as in the original data frame.

Notes

Conversion from/to tsfresh format requires pandas to be installed.

Examples

```
>>> import pandas as pd
>>> tsfresh_df = pd.DataFrame(columns=["id", "time", "a", "b"])
>>> tsfresh_df["id"] = [0, 0, 0]
>>> tsfresh_df["time"] = [0, 1, 2]
>>> tsfresh_df["a"] = [-1, 4, 7]
```

(continues on next page)

(continued from previous page)

```
>>> tsfresh_df["b"] = [8, -3, 2]
>>> tslearn_arr = from_tsfresh_dataset(tsfresh_df)
>>> tslearn_arr.shape
(1, 3, 2)
>>> tsfresh_df = pd.DataFrame(columns=["id", "time", "a"])
>>> tsfresh_df["id"] = [0, 0, 0, 1, 1]
>>> tsfresh_df["time"] = [0, 1, 2, 0, 1]
>>> tsfresh_df["a"] = [-1, 4, 7, 9, 1]
>>> tslearn_arr = from_tsfresh_dataset(tsfresh_df)
>>> tslearn_arr.shape
(2, 3, 1)
>>> tsfresh_df = numpy.random.randn(10, 1, 16)
>>> from_tsfresh_dataset(
...     tsfresh_df
... )  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
...
ValueError: X is not a valid input tsfresh array.
```

3.14.20 tslearn.utils.to_stumpy_dataset

tslearn.utils.**to_stumpy_dataset**(*X*)

Transform a tslearn-compatible dataset into a stumpy dataset.

Parameters

X: array, shape = (n_ts, sz, d) tslearn-formatted dataset to be cast to stumpy format

Returns

list of arrays of shape=(d, sz_i) if d > 1 or (sz_i,) otherwise stumpy-formatted dataset.

Examples

```
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> stumpy_arr = to_stumpy_dataset(tslearn_arr)
>>> len(stumpy_arr)
10
>>> stumpy_arr[0].shape
(16,)
>>> tslearn_arr = numpy.random.randn(10, 16, 2)
>>> stumpy_arr = to_stumpy_dataset(tslearn_arr)
>>> len(stumpy_arr)
10
>>> stumpy_arr[0].shape
(2, 16)
```

3.14.21 tslearn.utils.from_stumpy_dataset

tslearn.utils.**from_stumpy_dataset**(*X*)

Transform a stumpy-compatible dataset into a tslearn dataset.

Parameters

X: list of arrays of shapes (d, sz_i) if d > 1 or (sz_i,) otherwise stumpy-formatted dataset.

Returns

array, shape=(n_ts, sz, d), where sz is the maximum of all array lengths tslearn-formatted dataset

Examples

```
>>> stumpy_arr = [numpy.random.randn(10), numpy.random.randn(10)]
>>> tslearn_arr = from_stumpy_dataset(stumpy_arr)
>>> tslearn_arr.shape
(2, 10, 1)
>>> stumpy_arr = [numpy.random.randn(3, 10), numpy.random.randn(3, 5)]
>>> tslearn_arr = from_stumpy_dataset(stumpy_arr)
>>> tslearn_arr.shape
(2, 10, 3)
```

3.14.22 tslearn.utils.to_pyflux_dataset

`tslearn.utils.`**`to_pyflux_dataset`**`(X)`

Transform a tslearn-compatible dataset into a pyflux dataset.

Parameters

X: array, shape = (n_ts, sz, d), where n_ts=1 tslearn-formatted dataset to be cast to pyflux format

Returns

Pandas data-frame pyflux-formatted dataset (cf. link)

Notes

Conversion from/to pyflux format requires pandas to be installed.

Examples

```
>>> tslearn_arr = numpy.random.randn(1, 16, 1)
>>> pyflux_df = to_pyflux_dataset(tslearn_arr)
>>> pyflux_df.shape
(16, 1)
>>> pyflux_df.columns[0]
'dim_0'
>>> tslearn_arr = numpy.random.randn(1, 16, 2)
>>> pyflux_df = to_pyflux_dataset(tslearn_arr)
>>> pyflux_df.shape
(16, 2)
>>> pyflux_df.columns[1]
'dim_1'
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> to_pyflux_dataset(tslearn_arr)  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
...
ValueError: Array should be made of a single time series (10 here)
```

3.14.23 tslearn.utils.from_pyflux_dataset

tslearn.utils.**from_pyflux_dataset**(*X*)

Transform a pyflux-compatible dataset into a tslearn dataset.

Parameters

X: pandas data-frame pyflux-formatted dataset

Returns

array, shape=(n_ts, sz, d), where n_ts=1 tslearn-formatted dataset. Column order is kept the same as in the original data frame.

Notes

Conversion from/to pyflux format requires pandas to be installed.

Examples

```
>>> import pandas as pd
>>> pyflux_df = pd.DataFrame()
>>> pyflux_df["dim_0"] = numpy.random.rand(10)
>>> tslearn_arr = from_pyflux_dataset(pyflux_df)
>>> tslearn_arr.shape
(1, 10, 1)
>>> pyflux_df = pd.DataFrame()
>>> pyflux_df["dim_0"] = numpy.random.rand(10)
>>> pyflux_df["dim_1"] = numpy.random.rand(10)
>>> pyflux_df["dim_2"] = numpy.random.rand(10)
>>> tslearn_arr = from_pyflux_dataset(pyflux_df)
>>> tslearn_arr.shape
(1, 10, 3)
>>> pyflux_arr = numpy.random.randn(10, 1, 16)
>>> from_pyflux_dataset(
...     pyflux_arr
... )  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
...
ValueError: X is not a valid input pyflux array.
```

CHAPTER 4

Gallery of examples

4.1 Metrics

4.1.1 LB_Keogh

This example illustrates the principle of time series envelope and its relationship to the "LB_Keogh" lower bound [1].

The envelope of a time series consists of two time series such that the original time series is between the two time series. Denoting the original time series $X = (X_i)_{1 \leq i \leq n}$, the envelope of this time series is an ensemble of two time series of same length $L = (l_i)_{1 \leq i \leq n}$ and $U = (u_i)_{1 \leq i \leq n}$ such that for all $i \in \{1, \ldots, n\}$:

$$u_i = \max(x_{i-r}, \ldots, x_{i+r})$$
$$l_i = \min(x_{i-r}, \ldots, x_{i+r})$$

where r is the radius of the envelope.

The distance between a time series \$Q\$ and an envelope (L, U) is defined as:

$$LB_{Keogh}(Q, (L, U)) = \sqrt{\sum_{i=1}^{n} \begin{cases} (q_i - u_i)^2 & \text{if } q_i > u_i \\ (q_i - l_i)^2 & \text{if } q_i < l_i \\ 0 & \text{otherwise} \end{cases}}$$

So it is simply the Euclidean distance between Q and the envelope.

[1] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping". Knowledge and Information Systems, 7(3), 358-386 (2004).

-
-

```
# Author: Romain Tavenard
#         Johann Faouzi
# License: BSD 3 clause
```

(continues on next page)

(continued from previous page)

```
# sphinx_gallery_thumbnail_number = 2

import numpy
import matplotlib.pyplot as plt

from tslearn.generators import random_walks
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn import metrics

numpy.random.seed(0)
n_ts, sz, d = 2, 100, 1
dataset = random_walks(n_ts=n_ts, sz=sz, d=d)
scaler = TimeSeriesScalerMeanVariance(mu=0., std=1.)  # Rescale time series
dataset_scaled = scaler.fit_transform(dataset)

plt.figure(figsize=(14, 8))
envelope_down, envelope_up = metrics.lb_envelope(dataset_scaled[0], radius=3)
plt.plot(dataset_scaled[0, :, 0], "r-", label='First time series')
plt.plot(envelope_down[:, 0], "b-", label='Lower envelope')
plt.plot(envelope_up[:, 0], "g-", label='Upper envelope')
plt.legend()
plt.title('Envelope around a time series with radius=3')

plt.figure(figsize=(14, 8))
plt.plot(envelope_down[:, 0], "b-", label='Lower envelope')
plt.plot(envelope_up[:, 0], "g-", label='Upper envelope')
plt.plot(dataset_scaled[1, :, 0], "k-", label='Second time series')
plt.vlines(numpy.arange(sz), dataset_scaled[1, :, 0], numpy.clip(
    dataset_scaled[1, :, 0], envelope_down[:, 0], envelope_up[:, 0]),
           label='Distance', color='orange')
plt.legend()
lb_k_sim = metrics.lb_keogh(dataset_scaled[1],
                            envelope_candidate=(envelope_down, envelope_up))
plt.title('Distance between the second time series and \n'
          'the envelope = {:.4f}'.format(lb_k_sim))

plt.show()
```

Total running time of the script: (0 minutes 1.655 seconds)

4.1.2 sDTW multi path matching

This example illustrates how subsequent DTW can be used to find multiple matches of a sequence in a longer sequence.

A potential usecase is to identify the occurrence of certain events in continuous sensor signals. As one example Barth et al. [1] used this method to find stride in sensor recordings of gait.

The example demonstrates the use of the functions *subsequence_cost_matrix* and *subsequence_path* to manually calculate warping paths from multiple potential alignments. If you are only interested in finding the optimal alignment, you can directly use *dtw_subsequence_path*.

[1] Barth, et al. (2013): Subsequence dynamic time warping as a method for robust step segmentation using gyroscope signals of daily life activities, EMBS, https://doi.org/10.1109/EMBC.2013.6611104

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/tslearn/envs/latest/lib/python3.7/
↪importlib/_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may indicate␣
↪binary incompatibility. Expected 192 from C header, got 216 from PyObject
  return f(*args, **kwds)
Shape long sequence: (500, 1)
Shape short sequence: (100, 1)
```

```
# Author: Arne Kuederle
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy
from scipy.signal import find_peaks

from tslearn import metrics
from tslearn.generators import random_walks
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

numpy.random.seed(0)
n_ts, sz, d = 2, 100, 1
n_repeat = 5
dataset = random_walks(n_ts=n_ts, sz=sz, d=d)
scaler = TimeSeriesScalerMeanVariance(mu=0., std=1.)  # Rescale time series
dataset_scaled = scaler.fit_transform(dataset)

# We repeat the long sequence multiple times to generate multiple possible
# matches
long_sequence = numpy.tile(dataset_scaled[1], (n_repeat, 1))
short_sequence = dataset_scaled[0]

sz1 = len(long_sequence)
sz2 = len(short_sequence)

print('Shape long sequence: {}'.format(long_sequence.shape))
print('Shape short sequence: {}'.format(short_sequence.shape))

# Calculate the accumulated cost matrix
mat = metrics.subsequence_cost_matrix(short_sequence,
                                      long_sequence)

# Calculate cost function
cost_func = mat[-1, :]

# Identify potential matches in the cost function (parameters are tuned to
# fit this example)
potential_matches = find_peaks(-cost_func, distance=sz * 0.75, height=-50)[0]

# Calculate the optimal warping path starting from each of the identified
# minima
paths = [metrics.subsequence_path(mat, match) for match in
         potential_matches]
```

(continues on next page)

(continued from previous page)

```
plt.figure(1, figsize=(6 * n_repeat, 6))

# definitions for the axes
left, bottom = 0.01, 0.1
h_ts = 0.2
w_ts = h_ts / n_repeat
left_h = left + w_ts + 0.02
width = height = 0.65
bottom_h = bottom + height + 0.02

rect_s_y = [left, bottom, w_ts, height]
rect_gram = [left_h, bottom, width, height]
rect_s_x = [left_h, bottom_h, width, h_ts]

ax_gram = plt.axes(rect_gram)
ax_s_x = plt.axes(rect_s_x)
ax_s_y = plt.axes(rect_s_y)

ax_gram.imshow(numpy.sqrt(mat))
ax_gram.axis("off")
ax_gram.autoscale(False)

# Plot the paths
for path in paths:
    ax_gram.plot([j for (i, j) in path], [i for (i, j) in path], "w-",
                 linewidth=3.)

ax_s_x.plot(numpy.arange(sz1), long_sequence, "b-", linewidth=3.)
ax_s_x.axis("off")
ax_s_x.set_xlim((0, sz1 - 1))

ax_s_y.plot(- short_sequence, numpy.arange(sz2)[::-1], "b-", linewidth=3.)
ax_s_y.axis("off")
ax_s_y.set_ylim((0, sz2 - 1))

plt.show()
```

Total running time of the script: (0 minutes 2.777 seconds)

4.1.3 DTW computation

This example illustrates DTW computation between time series and plots the optimal alignment path [1].

The image represents cost matrix, that is the squared Euclidean distance for each time point between both time series, which are represented at the left and at the top of the cost matrix.

The optimal path, that is the path that minimizes the total cost to go from the first time point to the last one, is represented in white on the image.

[1] H. Sakoe and S. Chiba, “Dynamic programming algorithm optimization for spoken word recognition”. IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1), 43-49 (1978).

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/tslearn/checkouts/latest/docs/
↪examples/metrics/plot_dtw.py:105: UserWarning: This figure includes Axes that are␣
↪not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt

from tslearn.generators import random_walks
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn import metrics

numpy.random.seed(0)

s_x = numpy.array(
    [-0.790, -0.765, -0.734, -0.700, -0.668, -0.639, -0.612, -0.587, -0.564,
     -0.544, -0.529, -0.518, -0.509, -0.502, -0.494, -0.488, -0.482, -0.475,
     -0.472, -0.470, -0.465, -0.464, -0.461, -0.458, -0.459, -0.460, -0.459,
     -0.458, -0.448, -0.431, -0.408, -0.375, -0.333, -0.277, -0.196, -0.090,
     0.047, 0.220, 0.426, 0.671, 0.962, 1.300, 1.683, 2.096, 2.510, 2.895,
     3.219, 3.463, 3.621, 3.700, 3.713, 3.677, 3.606, 3.510, 3.400, 3.280,
     3.158, 3.038, 2.919, 2.801, 2.676, 2.538, 2.382, 2.206, 2.016, 1.821,
     1.627, 1.439, 1.260, 1.085, 0.917, 0.758, 0.608, 0.476, 0.361, 0.259,
     0.173, 0.096, 0.027, -0.032, -0.087, -0.137, -0.179, -0.221, -0.260,
     -0.293, -0.328, -0.359, -0.385, -0.413, -0.437, -0.458, -0.480, -0.498,
     -0.512, -0.526, -0.536, -0.544, -0.552, -0.556, -0.561, -0.565, -0.568,
     -0.570, -0.570, -0.566, -0.560, -0.549, -0.532, -0.510, -0.480, -0.443,
     -0.402, -0.357, -0.308, -0.256, -0.200, -0.139, -0.073, -0.003, 0.066,
     0.131, 0.186, 0.229, 0.259, 0.276, 0.280, 0.272, 0.256, 0.234, 0.209,
     0.186, 0.162, 0.139, 0.112, 0.081, 0.046, 0.008, -0.032, -0.071, -0.110,
     -0.147, -0.180, -0.210, -0.235, -0.256, -0.275, -0.292, -0.307, -0.320,
     -0.332, -0.344, -0.355, -0.363, -0.367, -0.364, -0.351, -0.330, -0.299,
     -0.260, -0.217, -0.172, -0.128, -0.091, -0.060, -0.036, -0.022, -0.016,
     -0.020, -0.037, -0.065, -0.104, -0.151, -0.201, -0.253, -0.302, -0.347,
     -0.388, -0.426, -0.460, -0.491, -0.517, -0.539, -0.558, -0.575, -0.588,
     -0.600, -0.606, -0.607, -0.604, -0.598, -0.589, -0.577, -0.558, -0.531,
     -0.496, -0.454, -0.410, -0.364, -0.318, -0.276, -0.237, -0.203, -0.176,
     -0.157, -0.145, -0.142, -0.145, -0.154, -0.168, -0.185, -0.206, -0.230,
     -0.256, -0.286, -0.318, -0.351, -0.383, -0.414, -0.442, -0.467, -0.489,
     -0.508, -0.523, -0.535, -0.544, -0.552, -0.557, -0.560, -0.560, -0.557,
     -0.551, -0.542, -0.531, -0.519, -0.507, -0.494, -0.484, -0.476, -0.469,
     -0.463, -0.456, -0.449, -0.442, -0.435, -0.431, -0.429, -0.430, -0.435,
     -0.442, -0.452, -0.465, -0.479, -0.493, -0.506, -0.517, -0.526, -0.535,
     -0.548, -0.567, -0.592, -0.622, -0.655, -0.690, -0.728, -0.764, -0.795,
     -0.815, -0.823, -0.821])

s_y1 = numpy.concatenate((s_x, s_x)).reshape((-1, 1))
s_y2 = numpy.concatenate((s_x, s_x[::-1])).reshape((-1, 1))
```

(continues on next page)

(continued from previous page)

```
sz = s_y1.shape[0]

path, sim = metrics.dtw_path(s_y1, s_y2)

plt.figure(1, figsize=(8, 8))

# definitions for the axes
left, bottom = 0.01, 0.1
w_ts = h_ts = 0.2
left_h = left + w_ts + 0.02
width = height = 0.65
bottom_h = bottom + height + 0.02

rect_s_y = [left, bottom, w_ts, height]
rect_gram = [left_h, bottom, width, height]
rect_s_x = [left_h, bottom_h, width, h_ts]

ax_gram = plt.axes(rect_gram)
ax_s_x = plt.axes(rect_s_x)
ax_s_y = plt.axes(rect_s_y)

mat = cdist(s_y1, s_y2)

ax_gram.imshow(mat, origin='lower')
ax_gram.axis("off")
ax_gram.autoscale(False)
ax_gram.plot([j for (i, j) in path], [i for (i, j) in path], "w-",
             linewidth=3.)

ax_s_x.plot(numpy.arange(sz), s_y2, "b-", linewidth=3.)
ax_s_x.axis("off")
ax_s_x.set_xlim((0, sz - 1))

ax_s_y.plot(- s_y1, numpy.arange(sz), "b-", linewidth=3.)
ax_s_y.axis("off")
ax_s_y.set_ylim((0, sz - 1))

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.987 seconds)

4.1.4 DTW computation with a custom distance metric

This example illustrates how to use the DTW computation of the optimal alignment path[1] on an user-defined distance matrix using *dtw_path_from_metric()*.

Left is the DTW of two angular time series using the length of the arc on the unit circle as a distance metric[2] and right is the DTW of two multidimensional boolean time series using hamming distance[3].

The images represent cost matrices, that is, on the left the length of the arc between each pair of angles on the unit circle and on the right the hamming distances between the multidimensional boolean arrays. In both cases, the corresponding

[1] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition". IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1), 43-49 (1978).

[2] Definition of the length of an arc on Wikipedia.

[3] See Hammig distance in Scipy's documentation.

time series are represented at the left and at the top of each cost matrix.

The optimal path, that is the path that minimizes the total user-defined cost from the first time point to the last one, is represented in white on the image.

-
-

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/tslearn/checkouts/latest/docs/
↪examples/metrics/plot_dtw_custom_metric.py:156: UserWarning: This figure includes␣
↪Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
```

```
# Author: Romain Fayat
# License: BSD 3 clause
# sphinx_gallery_thumbnail_number = 2

import numpy as np
from numpy import pi
from sklearn.metrics import pairwise_distances
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap

from tslearn.generators import random_walks
from tslearn import metrics
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

np.random.seed(0)
n_ts, sz = 2, 100


# Example 1 : Length of the arc between two angles on a circle
def arc_length(angle_1, angle_2, r=1.):
    """Length of the arc between two angles (in rad) on a circle of
    radius r.
    """
    # Compute the angle between the two inputs between 0 and 2*pi.
    theta = np.mod(angle_2 - angle_1, 2*pi)
    if theta > pi:
        theta = theta - 2 * pi
    # Return the length of the arc
    L = r * np.abs(theta)
    return(L)


dataset_1 = random_walks(n_ts=n_ts, sz=sz, d=1)
scaler = TimeSeriesScalerMeanVariance(mu=0., std=pi)  # Rescale the time series
dataset_scaled_1 = scaler.fit_transform(dataset_1)

# DTW using a function as the metric argument
path_1, sim_1 = metrics.dtw_path_from_metric(
```

(continues on next page)

(continued from previous page)

```
    dataset_scaled_1[0], dataset_scaled_1[1], metric=arc_length
)

# Example 2 : Hamming distance between 2 multi-dimensional boolean time series
rw = random_walks(n_ts=n_ts, sz=sz, d=15, std=.3)
dataset_2 = np.mod(np.floor(rw), 4) == 0

# DTW using one of the options of sklearn.metrics.pairwise_distances
path_2, sim_2 = metrics.dtw_path_from_metric(
    dataset_2[0], dataset_2[1], metric="hamming"
)

# Plots
# Compute the distance matrices for the plots
distances_1 = pairwise_distances(
    dataset_scaled_1[0], dataset_scaled_1[1], metric=arc_length
)
distances_2 = pairwise_distances(dataset_2[0], dataset_2[1], metric="hamming")

# Definitions for the axes
left, bottom = 0.01, 0.1
w_ts = h_ts = 0.2
left_h = left + w_ts + 0.02
width = height = 0.65
bottom_h = bottom + height + 0.02

rect_s_y = [left, bottom, w_ts, height]
rect_dist = [left_h, bottom, width, height]
rect_s_x = [left_h, bottom_h, width, h_ts]

# Plot example 1
plt.figure(1, figsize=(6, 6))
ax_dist = plt.axes(rect_dist)
ax_s_x = plt.axes(rect_s_x)
ax_s_y = plt.axes(rect_s_y)

ax_dist.imshow(distances_1, origin='lower')
ax_dist.axis("off")
ax_dist.autoscale(False)
ax_dist.plot(*zip(*path_1), "w-", linewidth=3.)

ticks_location = [-pi, 0, pi]
ticks_labels = [r"$\bf-\pi$", r"$\bf0$", r"$\bf\pi$"]

ax_s_x.plot([0, sz - 1], [ticks_location]*2, "k--", alpha=.2)
ax_s_x.plot(np.arange(sz), dataset_scaled_1[1], "b-", linewidth=3.)
ax_s_x.set_xlim((0, sz - 1))
ax_s_x.axis("off")

ax_s_y.plot([ticks_location]*2, [0, sz - 1], "k--", alpha=.2)
ax_s_y.plot(-dataset_scaled_1[0], np.arange(sz), "b-", linewidth=3.)
ax_s_y.set_ylim((0, sz - 1))
ax_s_y.axis("off")

for loc, s in zip(ticks_location, ticks_labels):
    ax_s_x.text(0, loc, s, fontsize="large", color="grey",
                horizontalalignment="right", verticalalignment="center")
```

(continues on next page)

(continued from previous page)

```
    ax_s_y.text(-loc, 0, s, fontsize="large", color="grey",
                horizontalalignment="center", verticalalignment="top")

# Plot example 2
plt.figure(2, figsize=(6, 6))
ax_dist = plt.axes(rect_dist)
ax_s_x = plt.axes(rect_s_x)
ax_s_y = plt.axes(rect_s_y)

ax_dist.imshow(distances_2, origin='lower')
ax_dist.axis("off")
ax_dist.autoscale(False)
ax_dist.plot(*zip(*path_2), "w-", linewidth=3.)

colors = [(1, 1, 1), (0, 0, 1)]  # White -> Blue
cmap_name = 'white_blue'
cm = LinearSegmentedColormap.from_list(cmap_name, colors, N=2)

ax_s_x.imshow(dataset_2[1].T, aspect="auto", cmap=cm)
ax_s_x.axis("off")

ax_s_y.imshow(np.flip(dataset_2[0], axis=1), aspect="auto", cmap=cm)
ax_s_y.axis("off")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.927 seconds)

4.2 Nearest Neighbors

4.2.1 k-NN search

This example performs a k-Nearest-Neighbor search in a database of time series using DTW as a base metric.

To do so, we use the *tslearn.neighbors.KNeighborsTimeSeries* class which provides utilities for the k-Nearest-Neighbor algorithm for time series.

[1] Wikipedia entry for the k-nearest neighbors algorithm

[2] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition". IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1), 43-49 (1978).

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.neighbors import KNeighborsTimeSeries
from tslearn.datasets import CachedDatasets

seed = 0
```

(continues on next page)

(continued from previous page)

```
numpy.random.seed(seed)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")

n_queries = 2
n_neighbors = 4

knn = KNeighborsTimeSeries(n_neighbors=n_neighbors)
knn.fit(X_train)
ind = knn.kneighbors(X_test[:n_queries], return_distance=False)

plt.figure()
for idx_ts in range(n_queries):
    plt.subplot(n_neighbors + 1, n_queries, idx_ts + 1)
    plt.plot(X_test[idx_ts].ravel(), "k-")
    plt.xticks([])
    for rank_nn in range(n_neighbors):
        plt.subplot(n_neighbors + 1, n_queries,
                    idx_ts + (n_queries * (rank_nn + 1)) + 1)
        plt.plot(X_train[ind[idx_ts, rank_nn]].ravel(), "r-")
        plt.xticks([])


plt.suptitle("Queries (in black) and their nearest neighbors (red)")
plt.show()
```

Total running time of the script: (0 minutes 0.651 seconds)

4.2.2 Nearest neighbors

This example illustrates the use of nearest neighbor methods for database search and classification tasks.

The three-nearest neighbors of the time series from a test set are computed. Then, the predictive performance of a three-nearest neighbors classifier [1] is computed with three different metrics: Dynamic Time Warping [2], Euclidean distance and SAX-MINDIST [3].

[1] Wikipedia entry for the k-nearest neighbors algorithm

[2] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition". IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1), 43-49 (1978).

[3] J. Lin, E. Keogh, L. Wei and S. Lonardi, "Experiencing SAX: a novel symbolic representation of time series". Data Mining and Knowledge Discovery, 15(2), 107-144 (2007).

Out:

```
1. Nearest neighbour search
Computed nearest neighbor indices (wrt DTW)
 [[10 12  2]
 [ 0 13  5]
 [ 0  1 13]
 [ 0 11  5]
 [16 18 12]
 [ 3 17  9]
 [12  2 16]
 [ 7  3 17]
 [12  2 10]
 [12  2 18]
```

(continues on next page)

(continued from previous page)

```
 [12  8  2]
 [ 3 17  7]
 [18 19  2]
 [ 0 17 13]
 [ 9  3  7]
 [12  2  8]
 [ 3  7  9]
 [ 0  1 13]
 [18 10  2]
 [10 12  2]]
First nearest neighbor class: [0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 0 1 0 0 0]

2. Nearest neighbor classification using DTW
Correct classification rate: 1.0

3. Nearest neighbor classification using L2
Correct classification rate: 1.0

4. Nearest neighbor classification using SAX+MINDIST
Correct classification rate: 0.5
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
from sklearn.metrics import accuracy_score

from tslearn.generators import random_walk_blobs
from tslearn.preprocessing import TimeSeriesScalerMinMax, \
    TimeSeriesScalerMeanVariance
from tslearn.neighbors import KNeighborsTimeSeriesClassifier, \
    KNeighborsTimeSeries

numpy.random.seed(0)
n_ts_per_blob, sz, d, n_blobs = 20, 100, 1, 2

# Prepare data
X, y = random_walk_blobs(n_ts_per_blob=n_ts_per_blob,
                         sz=sz,
                         d=d,
                         n_blobs=n_blobs)
scaler = TimeSeriesScalerMinMax(value_range=(0., 1.))  # Rescale time series
X_scaled = scaler.fit_transform(X)

indices_shuffle = numpy.random.permutation(n_ts_per_blob * n_blobs)
X_shuffle = X_scaled[indices_shuffle]
y_shuffle = y[indices_shuffle]

X_train = X_shuffle[:n_ts_per_blob * n_blobs // 2]
X_test = X_shuffle[n_ts_per_blob * n_blobs // 2:]
y_train = y_shuffle[:n_ts_per_blob * n_blobs // 2]
```

(continues on next page)

(continued from previous page)

```
y_test = y_shuffle[n_ts_per_blob * n_blobs // 2:]

# Nearest neighbor search
knn = KNeighborsTimeSeries(n_neighbors=3, metric="dtw")
knn.fit(X_train, y_train)
dists, ind = knn.kneighbors(X_test)
print("1. Nearest neighbour search")
print("Computed nearest neighbor indices (wrt DTW)\n", ind)
print("First nearest neighbor class:", y_test[ind[:, 0]])

# Nearest neighbor classification
knn_clf = KNeighborsTimeSeriesClassifier(n_neighbors=3, metric="dtw")
knn_clf.fit(X_train, y_train)
predicted_labels = knn_clf.predict(X_test)
print("\n2. Nearest neighbor classification using DTW")
print("Correct classification rate:", accuracy_score(y_test, predicted_labels))

# Nearest neighbor classification with a different metric (Euclidean distance)
knn_clf = KNeighborsTimeSeriesClassifier(n_neighbors=3, metric="euclidean")
knn_clf.fit(X_train, y_train)
predicted_labels = knn_clf.predict(X_test)
print("\n3. Nearest neighbor classification using L2")
print("Correct classification rate:", accuracy_score(y_test, predicted_labels))

# Nearest neighbor classification based on SAX representation
metric_params = {'n_segments': 10, 'alphabet_size_avg': 5}
knn_clf = KNeighborsTimeSeriesClassifier(n_neighbors=3, metric="sax",
                                         metric_params=metric_params)
knn_clf.fit(X_train, y_train)
predicted_labels = knn_clf.predict(X_test)
print("\n4. Nearest neighbor classification using SAX+MINDIST")
print("Correct classification rate:", accuracy_score(y_test, predicted_labels))
```

Total running time of the script: (0 minutes 0.190 seconds)

4.2.3 Hyper-parameter tuning of a Pipeline with KNeighborsTimeSeriesClassifier

In this example, we demonstrate how it is possible to use the different algorithms of tslearn in combination with sklearn utilities, such as the *sklearn.pipeline.Pipeline* and *sklearn.model_selection.GridSearchCV*. In this specific example, we will tune two of the hyper-parameters of a *KNeighborsTimeSeriesClassifier*.

Out:

```
Performing hyper-parameter tuning of KNN classifier... Done!

Got the following accuracies on the test set for each fold:
|n_neighbors |  weights   |score_fold_1|score_fold_2|score_fold_3|
---------------------------------------------------------------------
|          5|     uniform|     0.64706|     0.82353|      0.6875|
|          5|    distance|     0.70588|     0.88235|      0.8125|
|         25|     uniform|     0.64706|     0.64706|       0.625|
|         25|    distance|     0.82353|     0.76471|      0.8125|
```

(continues on next page)

(continued from previous page)

```
Best parameter combination:
weights=distance, n_neighbors=5
```

```
# Author: Gilles Vandewiele
# License: BSD 3 clause

from tslearn.neighbors import KNeighborsTimeSeriesClassifier
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.datasets import CachedDatasets

from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.pipeline import Pipeline

import numpy as np

import matplotlib.pyplot as plt

# Our pipeline consists of two phases. First, data will be normalized using
# min-max normalization. Afterwards, it is fed to a KNN classifier. For the
# KNN classifier, we tune the n_neighbors and weights hyper-parameters.
n_splits = 3
pipeline = GridSearchCV(
    Pipeline([
            ('normalize', TimeSeriesScalerMinMax()),
            ('knn', KNeighborsTimeSeriesClassifier())
    ]),
    {'knn__n_neighbors': [5, 25], 'knn__weights': ['uniform', 'distance']},
    cv=StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)
)

X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Keep only timeseries of class 1, 2, 3
X_train = X_train[y_train > 0]
y_train = y_train[y_train > 0]

# Keep only the first 50 timeseries of both train and
# retain only a small amount of each of the timeseries
X_train, y_train = X_train[:50, 50:150], y_train[:50]

# Plot our timeseries
colors = ['g', 'b', 'r']
plt.figure()
for ts, label in zip(X_train, y_train):
    plt.plot(ts, c=colors[label - 2], alpha=0.5)
plt.title('The timeseries in the dataset')
plt.tight_layout()
plt.show()

# Fit our pipeline
print(end='Performing hyper-parameter tuning of KNN classifier... ')
```

(continues on next page)

(continued from previous page)

```
pipeline.fit(X_train, y_train)
results = pipeline.cv_results_

# Print each possible configuration parameter and the out-of-fold accuracies
print('Done!')
print()
print('Got the following accuracies on the test set for each fold:')

header_str = '|'
columns = ['n_neighbors', 'weights']
columns += ['score_fold_{}'.format(i + 1) for i in range(n_splits)]
for col in columns:
    header_str += '{:^12}|'.format(col)
print(header_str)
print('-'*(len(columns) * 13))

for i in range(len(results['params'])):
    s = '|'
    s += '{:>12}|'.format(results['params'][i]['knn__n_neighbors'])
    s += '{:>12}|'.format(results['params'][i]['knn__weights'])
    for k in range(n_splits):
        score = results['split{}_test_score'.format(k)][i]
        score = np.around(score, 5)
        s += '{:>12}|'.format(score)
    print(s.strip())

best_comb = np.argmax(results['mean_test_score'])
best_params = results['params'][best_comb]

print()
print('Best parameter combination:')
print('weights={}, n_neighbors={}'.format(best_params['knn__weights'],
                                          best_params['knn__n_neighbors']))
```

Total running time of the script: (0 minutes 1.349 seconds)

4.2.4 1-NN with SAX + MINDIST

This example presents a comparison between k-Nearest Neighbor runs with k=1. It compares the use of: * MINDIST (see [1]) on SAX representations of the data. * Euclidean distance on the raw values of the time series.

The comparison is based on test accuracy using several benchmark datasets.

[1] Lin, Jessica, et al. "Experiencing SAX: a novel symbolic representation of time series." Data Mining and knowledge discovery 15.2 (2007): 107-144.

Out:

```
|      dataset       | sax error  |  sax time  | eucl error | eucl time  |
-------------------------------------------------------------------------
|    SyntheticControl|        0.03|     8.53082|        0.12|      0.0506|
|           GunPoint|     0.20667|      1.2238|     0.08667|     0.01904|
|           FaceFour|     0.14773|     0.68783|     0.21591|     0.01231|
|         Lightning2|     0.19672|     1.87309|      0.2459|     0.01394|
|         Lightning7|     0.46575|     1.45747|     0.42466|     0.01449|
|              ECG200|        0.12|     1.24898|        0.12|     0.01718|
```

(continues on next page)

(continued from previous page)

```
|               Plane|     0.04762|     1.83046|      0.0381|     0.01877|
|                 Car|        0.35|     1.77023|     0.26667|     0.01354|
|                Beef|     0.53333|     0.28849|     0.33333|     0.00683|
|              Coffee|     0.46429|     0.22908|         0.0|     0.00618|
|           OliveOil|     0.83333|     0.47205|     0.13333|     0.00724|
-----------------------------------------------------------------------------
```

```
# Author: Gilles Vandewiele
# License: BSD 3 clause

import warnings
import time

import numpy
import matplotlib.pyplot as plt
from scipy.stats import norm

from tslearn.datasets import UCR_UEA_datasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn.neighbors import KNeighborsTimeSeriesClassifier

from sklearn.base import clone
from sklearn.metrics import pairwise_distances, accuracy_score
from sklearn.neighbors import KNeighborsClassifier


warnings.filterwarnings('ignore')


def print_table(accuracies, times):
    """Utility function to pretty print the obtained accuracies"""
    header_str = '|'
    header_str += '{:^20}|'.format('dataset')
    columns = ['sax error', 'sax time', 'eucl error', 'eucl time']
    for col in columns:
        header_str += '{:^12}|'.format(col)
    print(header_str)
    print('-'*(len(columns) * 13 + 22))

    for dataset in accuracies:
        acc_sax, acc_euclidean = accuracies[dataset]
        time_sax, time_euclidean = times[dataset]
        sax_error = numpy.around(1 - acc_sax, 5)
        eucl_error = numpy.around(1 - acc_euclidean, 5)
        time_sax = numpy.around(time_sax, 5)
        time_euclidean = numpy.around(time_euclidean, 5)
        s = '|'
        s += '{:>20}|'.format(dataset)
        s += '{:>12}|'.format(sax_error)
        s += '{:>12}|'.format(time_sax)
        s += '{:>12}|'.format(eucl_error)
```

(continues on next page)

(continued from previous page)

```
        s += '{:>12}|'.format(time_euclidean)
        print(s.strip())

    print('-'*(len(columns) * 13 + 22))


# Set seed
numpy.random.seed(0)

# Defining dataset and the number of segments
data_loader = UCR_UEA_datasets()
datasets = [
    ('SyntheticControl', 16),
    ('GunPoint', 64),
    ('FaceFour', 128),
    ('Lightning2', 256),
    ('Lightning7', 128),
    ('ECG200', 32),
    ('Plane', 64),
    ('Car', 256),
    ('Beef', 128),
    ('Coffee', 128),
    ('OliveOil', 256)
]

# We will compare the accuracies & execution times of 1-NN using:
# (i) MINDIST on SAX representations, and
# (ii) euclidean distance on raw values
knn_sax = KNeighborsTimeSeriesClassifier(n_neighbors=1, metric='sax')
knn_eucl = KNeighborsTimeSeriesClassifier(n_neighbors=1, metric='euclidean')

accuracies = {}
times = {}
for dataset, w in datasets:
    X_train, y_train, X_test, y_test = data_loader.load_dataset(dataset)

    ts_scaler = TimeSeriesScalerMeanVariance()
    X_train = ts_scaler.fit_transform(X_train)
    X_test = ts_scaler.fit_transform(X_test)

    # Fit 1-NN using SAX representation & MINDIST
    metric_params = {'n_segments': w, 'alphabet_size_avg': 10}
    knn_sax = clone(knn_sax).set_params(metric_params=metric_params)
    start = time.time()
    knn_sax.fit(X_train, y_train)
    acc_sax = accuracy_score(y_test, knn_sax.predict(X_test))
    time_sax = time.time() - start

    # Fit 1-NN using euclidean distance on raw values
    start = time.time()
    knn_eucl.fit(X_train, y_train)
    acc_euclidean = accuracy_score(y_test, knn_eucl.predict(X_test))
    time_euclidean = time.time() - start

    accuracies[dataset] = (acc_sax, acc_euclidean)
    times[dataset] = (time_sax, time_euclidean)
```

(continues on next page)

(continued from previous page)

```
print_table(accuracies, times)
```

Total running time of the script: (0 minutes 27.717 seconds)

4.3 Clustering and Barycenters

4.3.1 KShape

This example uses the KShape clustering method [1] that is based on cross-correlation to cluster time series.

[1] J. Paparrizos & L. Gravano. k-Shape: Efficient and Accurate Clustering of Time Series. SIGMOD 2015. pp. 1855-1870.

Out:

```
0.008 --> 0.006 --> 0.004 --> 0.004 --> 0.004 --> 0.003 --> 0.003 --> 0.003 --> 0.003␣
↪--> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 -->␣
↪0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.
↪002 --> 0.002 -->
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.clustering import KShape
from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

seed = 0
numpy.random.seed(seed)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
# Keep first 3 classes and 50 first time series
X_train = X_train[y_train < 4]
X_train = X_train[:50]
numpy.random.shuffle(X_train)
# For this method to operate properly, prior scaling is required
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train)
sz = X_train.shape[1]

# kShape clustering
ks = KShape(n_clusters=3, verbose=True, random_state=seed)
y_pred = ks.fit_predict(X_train)

plt.figure()
for yi in range(3):
    plt.subplot(3, 1, 1 + yi)
```

(continues on next page)

(continued from previous page)

```
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-", alpha=.2)
    plt.plot(ks.cluster_centers_[yi].ravel(), "r-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.title("Cluster %d" % (yi + 1))

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 1.863 seconds)

4.3.2 Kernel k-means

This example uses Global Alignment kernel (GAK, [1]) at the core of a kernel k-means algorithm [2] to perform time series clustering.

Note that, contrary to k-means, a centroid cannot be computed when using kernel k-means. However, one can still report cluster assignments, which is what is provided here: each subfigure represents the set of time series from the training set that were assigned to the considered cluster.

[1] M. Cuturi, "Fast global alignment kernels," ICML 2011.

[2] I. S. Dhillon, Y. Guan, B. Kulis. Kernel k-means, Spectral Clustering and Normalized Cuts. KDD 2004.

Out:

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 1275 out of 1275 | elapsed:    6.2s finished
Init 1
80.948 --> 70.106 --> 66.011 --> 63.422 --> 59.720 --> 58.005 --> 57.563 --> 57.563 --
↪>
Init 2
80.519 --> 70.023 --> 66.522 --> 65.914 --> 65.914 -->
Init 3
80.374 --> 67.064 --> 62.859 --> 62.220 --> 59.391 --> 59.391 -->
Init 4
77.700 --> 69.585 --> 67.474 --> 67.022 --> 66.104 --> 65.075 --> 63.516 --> 62.861 --
↪> 62.410 --> 61.166 --> 59.759 --> 59.759 -->
Init 5
79.246 --> 66.190 --> 63.040 --> 63.040 -->
Init 6
78.590 --> 68.315 --> 66.321 --> 65.633 --> 63.898 --> 63.898 -->
Init 7
75.299 --> 63.203 --> 59.963 --> 57.563 --> 57.563 -->
Init 8
76.876 --> 67.042 --> 66.764 --> 66.764 -->
Init 9
81.317 --> 69.313 --> 63.927 --> 61.124 --> 59.391 --> 59.391 -->
Init 10
79.317 --> 72.390 --> 70.197 --> 70.218 --> 70.218 -->
Init 11
78.202 --> 66.888 --> 60.961 --> 57.946 --> 57.387 --> 57.387 -->
Init 12
78.194 --> 67.992 --> 65.263 --> 63.436 --> 61.177 --> 57.799 --> 57.387 --> 57.387 --
↪>
```

(continues on next page)

(continued from previous page)

```
Init 13
77.553 --> 64.028 --> 64.008 --> 64.008 -->
Init 14
77.853 --> 62.815 --> 57.799 --> 57.387 --> 57.387 -->
Init 15
81.746 --> 67.617 --> 63.332 --> 62.827 --> 62.234 --> 58.470 --> 57.387 --> 57.387 --
↪>
Init 16
78.934 --> 69.153 --> 65.466 --> 63.619 --> 63.619 -->
Init 17
78.303 --> 65.546 --> 63.619 --> 63.619 -->
Init 18
77.760 --> 67.020 --> 66.729 --> 65.900 --> 65.900 -->
Init 19
79.795 --> 70.429 --> 69.098 --> 69.098 -->
Init 20
79.419 --> 67.908 --> 65.330 --> 63.388 --> 61.019 --> 58.186 --> 57.387 --> 57.387 --
↪>
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.clustering import KernelKMeans
from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

seed = 0
numpy.random.seed(seed)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
# Keep first 3 classes
X_train = X_train[y_train < 4]
numpy.random.shuffle(X_train)
# Keep only 50 time series
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train[:50])
sz = X_train.shape[1]

gak_km = KernelKMeans(n_clusters=3,
                      kernel="gak",
                      kernel_params={"sigma": "auto"},
                      n_init=20,
                      verbose=True,
                      random_state=seed)
y_pred = gak_km.fit_predict(X_train)

plt.figure()
for yi in range(3):
    plt.subplot(3, 1, 1 + yi)
    for xx in X_train[y_pred == yi]:
```

(continues on next page)

(continued from previous page)

```
        plt.plot(xx.ravel(), "k-", alpha=.2)
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.title("Cluster %d" % (yi + 1))

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 6.511 seconds)

4.3.3 Barycenters

This example shows three methods to compute barycenters of time series. For an overview over the available methods see the *tslearn.barycenters* module.

tslearn provides three methods for calculating barycenters for a given set of time series:

- *Euclidean barycenter* is simply the arithmetic mean for each individual point in time, minimizing the summed euclidean distance for each of them. As can be seen below, it is very different from the DTW-based methods and may often be inappropriate. However, it is the fastest of the methods shown.
- *DTW Barycenter Averaging (DBA)* is an iteratively refined barycenter, starting out with a (potentially) bad candidate and improving it until convergence criteria are met. The optimization can be accomplished with (a) expectation-maximization [1] and (b) stochastic subgradient descent [2]. Empirically, the latter "is [often] more stable and finds better solutions in shorter time" [2].
- *Soft-DTW barycenter* uses a differentiable loss function to iteratively find a barycenter [3]. The method itself and the parameter $\gamma = 1.0$ is described in more detail in the section on *DTW*. There is also a dedicated *example* available.

[1] F. Petitjean, A. Ketterlin & P. Gancarski. A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition, Elsevier, 2011, Vol. 44, Num. 3, pp. 678-693.

[2] D. Schultz & B. Jain. Nonsmooth Analysis and Subgradient Methods for Averaging in Dynamic Time Warping Spaces. Pattern Recognition, 74, 340-358.

[3] M. Cuturi & M. Blondel. Soft-DTW: a Differentiable Loss Function for Time-Series. ICML 2017.

```
# Author: Romain Tavenard, Felix Divo
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.barycenters import \
    euclidean_barycenter, \
    dtw_barycenter_averaging, \
    dtw_barycenter_averaging_subgradient, \
    softdtw_barycenter
from tslearn.datasets import CachedDatasets

# fetch the example data set
numpy.random.seed(0)
X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")
X = X_train[y_train == 2]
```

(continues on next page)

(continued from previous page)

```
length_of_sequence = X.shape[1]


def plot_helper(barycenter):
    # plot all points of the data set
    for series in X:
        plt.plot(series.ravel(), "k-", alpha=.2)
    # plot the given barycenter of them
    plt.plot(barycenter.ravel(), "r-", linewidth=2)


# plot the four variants with the same number of iterations and a tolerance of
# 1e-3 where applicable
ax1 = plt.subplot(4, 1, 1)
plt.title("Euclidean barycenter")
plot_helper(euclidean_barycenter(X))

plt.subplot(4, 1, 2, sharex=ax1)
plt.title("DBA (vectorized version of Petitjean's EM)")
plot_helper(dtw_barycenter_averaging(X, max_iter=50, tol=1e-3))

plt.subplot(4, 1, 3, sharex=ax1)
plt.title("DBA (subgradient descent approach)")
plot_helper(dtw_barycenter_averaging_subgradient(X, max_iter=50, tol=1e-3))

plt.subplot(4, 1, 4, sharex=ax1)
plt.title("Soft-DTW barycenter ($\gamma$=1.0)")
plot_helper(softdtw_barycenter(X, gamma=1., max_iter=50, tol=1e-3))

# clip the axes for better readability
ax1.set_xlim([0, length_of_sequence])

# show the plot(s)
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 5.651 seconds)

4.3.4 Soft-DTW weighted barycenters

This example presents the weighted Soft-DTW time series barycenter method.

Soft-DTW [1] is a differentiable loss function for Dynamic Time Warping, allowing for the use of gradient-based algorithms. The barycenter corresponds to the time series that minimizes the sum of the distances between that time series and all the time series from a dataset. It is thus an optimization problem and having a differentiable loss function makes find the solution much easier.

In this example, we consider four time series X_0, X_1, X_2 and X_3 from four different classes in the Trace dataset. We compute the barycenters for different sets of weights and plot them. The closer to a time series the barycenter is, the higher the weight for this time series is.

[1] M. Cuturi and M. Blondel, "Soft-DTW: a Differentiable Loss Function for Time-Series". International Conference on Machine Learning, 2017.

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt
import matplotlib.colors

from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.barycenters import softdtw_barycenter
from tslearn.datasets import CachedDatasets


def row_col(position, n_cols=5):
    idx_row = (position - 1) // n_cols
    idx_col = position - n_cols * idx_row - 1
    return idx_row, idx_col


def get_color(weights):
    baselines = numpy.zeros((4, 3))
    weights = numpy.array(weights).reshape(1, 4)
    for i, c in enumerate(["r", "g", "b", "y"]):
        baselines[i] = matplotlib.colors.ColorConverter().to_rgb(c)
    return numpy.dot(weights, baselines).ravel()


numpy.random.seed(0)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
X_out = numpy.empty((4, X_train.shape[1], X_train.shape[2]))

plt.figure()
for i in range(4):
    X_out[i] = X_train[y_train == (i + 1)][0]
X_out = TimeSeriesScalerMinMax().fit_transform(X_out)

for i, pos in enumerate([1, 5, 21, 25]):
    plt.subplot(5, 5, pos)
    w = [0.] * 4
    w[i] = 1.
    plt.plot(X_out[i].ravel(),
             color=matplotlib.colors.rgb2hex(get_color(w)),
             linewidth=2)
    plt.text(X_out[i].shape[0], 0., "$X_%d$" % i,
             horizontalalignment="right",
             verticalalignment="baseline",
             fontsize=24)
    plt.xticks([])
    plt.yticks([])

for pos in range(2, 25):
    if pos in [1, 5, 21, 25]:
        continue
    plt.subplot(5, 5, pos)
    idxr, idxc = row_col(pos, 5)
    w = numpy.array([0.] * 4)
    w[0] = (4 - idxr) * (4 - idxc) / 16
    w[1] = (4 - idxr) * idxc / 16
```

(continues on next page)

(continued from previous page)

```
    w[2] = idxr * (4 - idxc) / 16
    w[3] = idxr * idxc / 16
    plt.plot(softdtw_barycenter(X=X_out, weights=w).ravel(),
             color=matplotlib.colors.rgb2hex(get_color(w)),
             linewidth=2)
    plt.xticks([])
    plt.yticks([])


plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 6.467 seconds)

4.3.5 k-means

This example uses k-means clustering for time series. Three variants of the algorithm are available: standard Euclidean k-means, DBA-k-means (for DTW Barycenter Averaging [1]) and Soft-DTW k-means [2].

In the figure below, each row corresponds to the result of a different clustering. In a row, each sub-figure corresponds to a cluster. It represents the set of time series from the training set that were assigned to the considered cluster (in black) as well as the barycenter of the cluster (in red).

A note on pre-processing

In this example, time series are preprocessed using *TimeSeriesScalerMeanVariance*. This scaler is such that each output time series has zero mean and unit variance. The assumption here is that the range of a given time series is uninformative and one only wants to compare shapes in an amplitude-invariant manner (when time series are multivariate, this also rescales all modalities such that there will not be a single modality responsible for a large part of the variance). This means that one cannot scale barycenters back to data range because each time series is scaled independently and there is hence no such thing as an overall data range.

[1] F. Petitjean, A. Ketterlin & P. Gancarski. A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition, Elsevier, 2011, Vol. 44, Num. 3, pp. 678-693 [2] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.

Out:

```
Euclidean k-means
16.434 --> 9.437 --> 9.437 -->
DBA k-means
Init 1
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  50 out of  50 | elapsed:    0.0s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
0.637 --> [Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent␣
↪workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
```

(continues on next page)

(continued from previous page)

```
0.458 --> [Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent␣
↪workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
0.458 -->
Init 2
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  50 out of  50 | elapsed:    0.0s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
0.826 --> [Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent␣
↪workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
0.525 --> [Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent␣
↪workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
0.477 --> [Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent␣
↪workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
0.472 --> [Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent␣
↪workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
0.472 -->
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed:    0.0s finished
Soft-DTW k-means
0.472 --> 0.144 --> 0.142 --> 0.143 --> 0.142 --> 0.143 --> 0.142 --> 0.143 --> 0.142␣
↪--> 0.142 --> 0.142 --> 0.142 --> 0.142 --> 0.142 --> 0.142 --> 0.142 --> 0.142 -->␣
↪0.142 --> 0.142 --> 0.142 --> 0.142 -->
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.clustering import TimeSeriesKMeans
from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance, \
    TimeSeriesResampler

seed = 0
numpy.random.seed(seed)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
X_train = X_train[y_train < 4]  # Keep first 3 classes
numpy.random.shuffle(X_train)
# Keep only 50 time series
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train[:50])
```

(continues on next page)

(continued from previous page)

```
# Make time series shorter
X_train = TimeSeriesResampler(sz=40).fit_transform(X_train)
sz = X_train.shape[1]

# Euclidean k-means
print("Euclidean k-means")
km = TimeSeriesKMeans(n_clusters=3, verbose=True, random_state=seed)
y_pred = km.fit_predict(X_train)

plt.figure()
for yi in range(3):
    plt.subplot(3, 3, yi + 1)
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-", alpha=.2)
    plt.plot(km.cluster_centers_[yi].ravel(), "r-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.text(0.55, 0.85,'Cluster %d' % (yi + 1),
             transform=plt.gca().transAxes)
    if yi == 1:
        plt.title("Euclidean $k$-means")

# DBA-k-means
print("DBA k-means")
dba_km = TimeSeriesKMeans(n_clusters=3,
                          n_init=2,
                          metric="dtw",
                          verbose=True,
                          max_iter_barycenter=10,
                          random_state=seed)
y_pred = dba_km.fit_predict(X_train)

for yi in range(3):
    plt.subplot(3, 3, 4 + yi)
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-", alpha=.2)
    plt.plot(dba_km.cluster_centers_[yi].ravel(), "r-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.text(0.55, 0.85,'Cluster %d' % (yi + 1),
             transform=plt.gca().transAxes)
    if yi == 1:
        plt.title("DBA $k$-means")

# Soft-DTW-k-means
print("Soft-DTW k-means")
sdtw_km = TimeSeriesKMeans(n_clusters=3,
                           metric="softdtw",
                           metric_params={"gamma": .01},
                           verbose=True,
                           random_state=seed)
y_pred = sdtw_km.fit_predict(X_train)

for yi in range(3):
    plt.subplot(3, 3, 7 + yi)
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-", alpha=.2)
```

(continues on next page)

(continued from previous page)

```
    plt.plot(sdtw_km.cluster_centers_[yi].ravel(), "r-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.text(0.55, 0.85,'Cluster %d' % (yi + 1),
             transform=plt.gca().transAxes)
    if yi == 1:
        plt.title("Soft-DTW $k$-means")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 5.535 seconds)

4.4 Classification

4.4.1 SVM and GAK

This example illustrates the use of the global alignment kernel (GAK) for support vector classification.

This metric is defined in the *tslearn.metrics* module and explained in details in [1].

In this example, a *TimeSeriesSVC* model that uses GAK as kernel is fit and the support vectors for each class are reported.

[1] M. Cuturi, "Fast global alignment kernels," ICML 2011.

Out:

```
Correct classification rate: 1.0
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.svm import TimeSeriesSVC

numpy.random.seed(0)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
X_train = TimeSeriesScalerMinMax().fit_transform(X_train)
X_test = TimeSeriesScalerMinMax().fit_transform(X_test)

clf = TimeSeriesSVC(kernel="gak", gamma=.1)
clf.fit(X_train, y_train)
print("Correct classification rate:", clf.score(X_test, y_test))
```

(continues on next page)

(continued from previous page)

```
n_classes = len(set(y_train))

plt.figure()
support_vectors = clf.support_vectors_
for i, cl in enumerate(set(y_train)):
    plt.subplot(n_classes, 1, i + 1)
    plt.title("Support vectors for class %d" % cl)
    for ts in support_vectors[i]:
        plt.plot(ts.ravel())

plt.tight_layout()
plt.show()
```

Total running time of the script: (1 minutes 19.958 seconds)

4.4.2 Learning Shapelets

This example illustrates how the "Learning Shapelets" method can quickly find a set of shapelets that results in excellent predictive performance when used for a shapelet transform.

More information on the method can be found at: http://fs.ismll.de/publicspace/LearningShapelets/.

-
-

Out:

```
Correct classification rate: 1.0
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
from sklearn.metrics import accuracy_score
import tensorflow as tf
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.shapelets import LearningShapelets, \
    grabocka_params_to_shapelet_size_dict
from tslearn.utils import ts_size

# Set seed for determinism
numpy.random.seed(0)

# Load the Trace dataset
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
```

(continues on next page)

(continued from previous page)

```
# Normalize each of the timeseries in the Trace dataset
X_train = TimeSeriesScalerMinMax().fit_transform(X_train)
X_test = TimeSeriesScalerMinMax().fit_transform(X_test)

# Get statistics of the dataset
n_ts, ts_sz = X_train.shape[:2]
n_classes = len(set(y_train))

# Set the number of shapelets per size as done in the original paper
shapelet_sizes = grabocka_params_to_shapelet_size_dict(n_ts=n_ts,
                                                       ts_sz=ts_sz,
                                                       n_classes=n_classes,
                                                       l=0.1,
                                                       r=1)

# Define the model using parameters provided by the authors (except that we
# use fewer iterations here)
shp_clf = LearningShapelets(n_shapelets_per_size=shapelet_sizes,
                            optimizer=tf.optimizers.Adam(.01),
                            batch_size=16,
                            weight_regularizer=.01,
                            max_iter=200,
                            random_state=42,
                            verbose=0)
shp_clf.fit(X_train, y_train)

# Make predictions and calculate accuracy score
pred_labels = shp_clf.predict(X_test)
print("Correct classification rate:", accuracy_score(y_test, pred_labels))

# Plot the different discovered shapelets
plt.figure()
for i, sz in enumerate(shapelet_sizes.keys()):
    plt.subplot(len(shapelet_sizes), 1, i + 1)
    plt.title("%d shapelets of size %d" % (shapelet_sizes[sz], sz))
    for shp in shp_clf.shapelets_:
        if ts_size(shp) == sz:
            plt.plot(shp.ravel())
    plt.xlim([0, max(shapelet_sizes.keys()) - 1])

plt.tight_layout()
plt.show()

# The loss history is accessible via the `model_` that is a keras model
plt.figure()
plt.plot(numpy.arange(1, shp_clf.n_iter_ + 1), shp_clf.history_["loss"])
plt.title("Evolution of cross-entropy loss during training")
plt.xlabel("Epochs")
plt.show()
```

Total running time of the script: (0 minutes 6.854 seconds)

4.4.3 Early Classification

This example presents the concept of early classification.

Early classifiers are implemented in the *tslearn.early_classification* module and in this example we use the method from [1].

[1] A. Dachraoui, A. Bondu & A. Cornuejols. Early classification of time series as a non myopic sequential decision making problem. ECML/PKDD 2015

```
# Author: Romain Tavenard
# License: BSD 3 clause
# sphinx_gallery_thumbnail_number = 2

import numpy
import matplotlib.pyplot as plt

from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn.early_classification import NonMyopicEarlyClassifier
from tslearn.datasets import UCR_UEA_datasets

def plot_partial(time_series, t, y_true=0, y_pred=0, color="k"):
    plt.plot(time_series[:t+1].ravel(), color=color, linewidth=1.5)
    plt.plot(numpy.arange(t+1, time_series.shape[0]),
             time_series[t+1:].ravel(),
             linestyle="dashed", color=color, linewidth=1.5)
    plt.axvline(x=t, color=color, linewidth=1.5)
    plt.text(x=t - 20, y=time_series.max() - .25, s="Prediction time")
    plt.title(
        "Sample of class {} predicted as class {}".format(y_true, y_pred)
    )
    plt.xlim(0, time_series.shape[0] - 1)
```

Data loading and visualization

```
numpy.random.seed(0)
X_train, y_train, X_test, y_test = UCR_UEA_datasets().load_dataset("ECG200")

# Scale time series
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train)
X_test = TimeSeriesScalerMeanVariance().fit_transform(X_test)

size = X_train.shape[1]
n_classes = len(set(y_train))

plt.figure()
for i, cl in enumerate(set(y_train)):
    plt.subplot(n_classes, 1, i + 1)
    for ts in X_train[y_train == cl]:
        plt.plot(ts.ravel(), color="orange" if cl > 0 else "blue", alpha=.3)
    plt.xlim(0, size - 1)
plt.suptitle("Training time series")
plt.show()
```

Model fitting

As observed in the following figure, the optimal classification time as estimated by *NonMyopicEarlyClassifier* is data-dependent.

```
early_clf = NonMyopicEarlyClassifier(n_clusters=3,
                                     cost_time_parameter=1e-3,
                                     lamb=1e2,
                                     random_state=0)
early_clf.fit(X_train, y_train)

preds, times = early_clf.predict_class_and_earliness(X_test)

plt.figure()
plt.subplot(2, 1, 1)
ts_idx = 0
t = times[ts_idx]
plot_partial(X_test[ts_idx], t, y_test[ts_idx], preds[ts_idx], color="orange")


plt.subplot(2, 1, 2)
ts_idx = 9
t = times[ts_idx]
plot_partial(X_test[ts_idx], t, y_test[ts_idx], preds[ts_idx], color="blue")
plt.tight_layout()
plt.show()
```

Earliness-Accuracy trade-off

The trade-off between earliness and accuracy is controlled via `cost_time_parameter`.

```
plt.figure()
hatches = ["///", "\\\\\\", "*"]
for i, cost_t in enumerate([1e-4, 1e-3, 1e-2]):
    early_clf.set_params(cost_time_parameter=cost_t)
    early_clf.fit(X_train, y_train)
    preds, times = early_clf.predict_class_and_earliness(X_test)
    plt.hist(times,
             alpha=.5, hatch=hatches[i],
             density=True,
             label="$\\alpha={}$".format(cost_t),
             bins=numpy.arange(0, size, 5))
plt.legend(loc="upper right")
plt.xlim(0, size - 1)
plt.xlabel("Prediction times")
plt.title("Impact of cost_time_parameter ($\\alpha$)")
plt.show()
```

Total running time of the script: (0 minutes 8.460 seconds)

4.4.4 Aligning discovered shapelets with timeseries

This example illustrates the use of the "Learning Shapelets" method in order to learn a collection of shapelets that linearly separates the timeseries. In this example, we will extract a single shapelet in order to distinguish between two classes of the "Trace" dataset. Afterwards, we show how our time series can be transformed to distances by aligning the shapelets along each of the time series. This alignment is performed by shifting the smaller shapelet across the longer time series and taking the minimal pointwise distance.

More information on the method can be found at: http://fs.ismll.de/publicspace/LearningShapelets/.

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.shapelets import LearningShapelets, \
    grabocka_params_to_shapelet_size_dict
from tensorflow.keras.optimizers import Adam

# Set a seed to ensure determinism
numpy.random.seed(42)

# Load the Trace dataset
X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Filter out classes 2 and 4
mask = numpy.isin(y_train, [1, 3])
X_train = X_train[mask]
y_train = y_train[mask]

# Normalize the time series
X_train = TimeSeriesScalerMinMax().fit_transform(X_train)

# Get statistics of the dataset
n_ts, ts_sz = X_train.shape[:2]
n_classes = len(set(y_train))

# We will extract 1 shapelet and align it with a time series
shapelet_sizes = {20: 1}

# Define the model and fit it using the training data
shp_clf = LearningShapelets(n_shapelets_per_size=shapelet_sizes,
                            weight_regularizer=0.001,
                            optimizer=Adam(lr=0.01),
                            max_iter=250,
                            verbose=0,
                            scale=False,
                            random_state=42)
shp_clf.fit(X_train, y_train)

# Get the number of extracted shapelets, the (minimal) distances from
# each of the timeseries to each of the shapelets, and the corresponding
# locations (index) where the minimal distance was found
n_shapelets = sum(shapelet_sizes.values())
distances = shp_clf.transform(X_train)
predicted_locations = shp_clf.locate(X_train)

f, ax = plt.subplots(2, 1, sharex=True)

# Plot the shapelet and align it on the best matched time series. The optimizer
# will often enlarge the shapelet to create a larger gap between the distances
```

(continues on next page)

(continued from previous page)

```
# of both classes. We therefore normalize the shapelet again before plotting.
test_ts_id = numpy.argmin(numpy.sum(distances, axis=1))
shap = shp_clf.shapelets_[0]
shap = TimeSeriesScalerMinMax().fit_transform(shap.reshape(1, -1, 1)).flatten()
pos = predicted_locations[test_ts_id, 0]
ax[0].plot(X_train[test_ts_id].ravel())
ax[0].plot(numpy.arange(pos, pos + len(shap)), shap, linewidth=2)
ax[0].axvline(pos, color='k', linestyle='--', alpha=0.25)
ax[0].set_title("The aligned extracted shapelet")

# We calculate the distances from the shapelet to the timeseries ourselves.
distances = []
time_series = X_train[test_ts_id].ravel()
for i in range(len(time_series) - len(shap)):
    distances.append(numpy.linalg.norm(time_series[i:i+len(shap)] - shap))
ax[1].plot(distances)
ax[1].axvline(numpy.argmin(distances), color='k', linestyle='--', alpha=0.25)
ax[1].set_title('The distances between the time series and the shapelet')

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 3.386 seconds)

4.4.5 Learning Shapelets: decision boundaries in 2D distance space

This example illustrates the use of the "Learning Shapelets" method in order to learn a collection of shapelets that linearly separates the timeseries. In this example, we will extract two shapelets which are then used to transform our input time series in a two-dimensional space, which is called the shapelet-transform space in the related literature. Moreover, we plot the decision boundaries of our classifier for each of the different classes.

More information on the method can be found at: http://fs.ismll.de/publicspace/LearningShapelets/.

```
# Author: Gilles Vandewiele
# License: BSD 3 clause

import numpy
from matplotlib import cm
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.shapelets import LearningShapelets
from tensorflow.keras.optimizers import Adam

# Set a seed to ensure determinism
numpy.random.seed(42)

# Load the Trace dataset
X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Normalize the time series
X_train = TimeSeriesScalerMinMax().fit_transform(X_train)
```

(continues on next page)

(continued from previous page)

```
# Get statistics of the dataset
n_ts, ts_sz = X_train.shape[:2]
n_classes = len(set(y_train))

# We will extract 2 shapelets and align them with the time series
shapelet_sizes = {20: 2}

# Define the model and fit it using the training data
shp_clf = LearningShapelets(n_shapelets_per_size=shapelet_sizes,
                            weight_regularizer=0.0001,
                            optimizer=Adam(lr=0.01),
                            max_iter=300,
                            verbose=0,
                            scale=False,
                            random_state=42)
shp_clf.fit(X_train, y_train)

# We will plot our distances in a 2D space
distances = shp_clf.transform(X_train).reshape((-1, 2))
weights, biases = shp_clf.get_weights('classification')

# Create a grid for our two shapelets on the left and distances on the right
viridis = cm.get_cmap('viridis', 4)
fig = plt.figure(constrained_layout=True)
gs = fig.add_gridspec(3, 9)
fig_ax1 = fig.add_subplot(gs[0, :2])
fig_ax2 = fig.add_subplot(gs[0, 2:4])
fig_ax3a = fig.add_subplot(gs[1, :2])
fig_ax3b = fig.add_subplot(gs[1, 2:4])
fig_ax3c = fig.add_subplot(gs[2, :2])
fig_ax3d = fig.add_subplot(gs[2, 2:4])
fig_ax4 = fig.add_subplot(gs[:, 4:])

# Plot our two shapelets on the left side
fig_ax1.plot(shp_clf.shapelets_[0])
fig_ax1.set_title('Shapelet $\mathbf{s}_1$')

fig_ax2.plot(shp_clf.shapelets_[1])
fig_ax2.set_title('Shapelet $\mathbf{s}_2$')

# Create the time series of each class
for i, subfig in enumerate([fig_ax3a, fig_ax3b, fig_ax3c, fig_ax3d]):
    for k, ts in enumerate(X_train[y_train == i + 1]):
        subfig.plot(ts.flatten(), c=viridis(i / 3), alpha=0.25)
        subfig.set_title('Class {}'.format(i + 1))
fig.text(x=.15, y=.02, s='Input time series', fontsize=12)

# Create a scatter plot of the 2D distances for the time series of each class.
for i, y in enumerate(numpy.unique(y_train)):
    fig_ax4.scatter(distances[y_train == y][:, 0],
                    distances[y_train == y][:, 1],
                    c=[viridis(i / 3)] * numpy.sum(y_train == y),
                    edgecolors='k',
                    label='Class {}'.format(y))

# Create a meshgrid of the decision boundaries
xmin = numpy.min(distances[:, 0]) - 0.1
```

(continues on next page)

(continued from previous page)

```
xmax = numpy.max(distances[:, 0]) + 0.1
ymin = numpy.min(distances[:, 1]) - 0.1
ymax = numpy.max(distances[:, 1]) + 0.1
xx, yy = numpy.meshgrid(numpy.arange(xmin, xmax, (xmax - xmin)/200),
                        numpy.arange(ymin, ymax, (ymax - ymin)/200))
Z = []
for x, y in numpy.c_[xx.ravel(), yy.ravel()]:
    Z.append(numpy.argmax([biases[i] + weights[0][i]*x + weights[1][i]*y
                           for i in range(4)]))
Z = numpy.array(Z).reshape(xx.shape)
cs = fig_ax4.contourf(xx, yy, Z / 3, cmap=viridis, alpha=0.25)

fig_ax4.legend()
fig_ax4.set_xlabel('$d(\mathbf{x}, \mathbf{s}_1)$')
fig_ax4.set_ylabel('$d(\mathbf{x}, \mathbf{s}_2)$')
fig_ax4.set_xlim((xmin, xmax))
fig_ax4.set_ylim((ymin, ymax))
fig_ax4.set_title('Distance transformed time series')
plt.show()
```

Total running time of the script: (0 minutes 8.554 seconds)

4.5 Miscellaneous

4.5.1 Model Persistence

Many tslearn models can be saved to disk and used for predictions at a later time. This can be particularly useful when a model takes a long time to train.

Available formats: hdf5, json, pickle

Save a model to disk:

```
model.to_<format>
```

Load a model from disk:

```
model.from_<format>
```

Basic usage

```
# Instantiate a model
model = ModelClass(<hyper-parameters>)

# Train the model
model.fit(X_train)

# Save the model to disk
model.to_hdf5('./trained_model.hdf5')

# Load model from disk
model.from_hdf5('./trained_mode.hdf5')

# Make predictions
y = model.predict(X_test)
```

Note: For the following models the training data are saved to disk and may result in a large model file if the trainig dataset is large: `KNeighborsTimeSeries`, `KNeighborsTimeSeriesClassifier`, and `KernelKMeans`

Out:

```
0.009 --> 0.009 --> 0.008 --> 0.008 --> 0.008 --> 0.007 --> 0.007 --> 0.006 --> 0.005␣
↪--> 0.005 --> 0.005 --> 0.005 --> 0.004 --> 0.004 --> 0.004 --> 0.004 --> 0.004 -->␣
↪0.004 --> 0.003 --> 0.003 --> 0.003 --> 0.003 --> 0.003 --> 0.003 --> 0.003 --> 0.
↪003 --> 0.003 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002␣
↪--> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 -->␣
↪0.002 --> 0.002 --> 0.002 --> 0.002 -->
```

```
# Example using KShape

import numpy
import matplotlib.pyplot as plt

from tslearn.clustering import KShape
from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

seed = 0
numpy.random.seed(seed)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")

# Keep first 3 classes
X_train = X_train[y_train < 4]
numpy.random.shuffle(X_train)
# Keep only 50 time series
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train[:50])
sz = X_train.shape[1]

# Instantiate k-Shape model
ks = KShape(n_clusters=3, verbose=True, random_state=seed)

# Train
ks.fit(X_train)

# Save model
ks.to_hdf5('./ks_trained.hdf5')

# Load model
trained_ks = KShape.from_hdf5('./ks_trained.hdf5')

# Use loaded model to make predictions
y_pred = trained_ks.predict(X_train)

plt.figure()
for yi in range(3):
```

(continues on next page)

(continued from previous page)

```
    plt.subplot(3, 1, 1 + yi)
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-", alpha=.2)
    plt.plot(ks.cluster_centers_[yi].ravel(), "r-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.title("Cluster %d" % (yi + 1))

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 3.022 seconds)

4.5.2 PAA and SAX features

This example presents a comparison between PAA [1], SAX [2] and 1d-SAX [3] features.

PAA (Piecewise Aggregate Approximation) corresponds to a downsampling of the original time series and, in each segment (segments have fixed size), the mean value is retained.

SAX (Symbolic Aggregate approXimation) builds upon PAA by quantizing the mean value. Quantization boundaries are computed for all symbols to be equiprobable, under a standard normal distribution assumption.

Finally, 1d-SAX is an extension of SAX in which each segment is represented by an affine function (2 parameters per segment are hence quantized: slope and mean value).

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[2] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[3] S. Malinowski, T. Guyet, R. Quiniou, R. Tavenard. 1d-SAX: a Novel Symbolic Representation for Time Series. IDA 2013.

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.generators import random_walks
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn.piecewise import PiecewiseAggregateApproximation
from tslearn.piecewise import SymbolicAggregateApproximation, \
    OneD_SymbolicAggregateApproximation

numpy.random.seed(0)
# Generate a random walk time series
n_ts, sz, d = 1, 100, 1
dataset = random_walks(n_ts=n_ts, sz=sz, d=d)
scaler = TimeSeriesScalerMeanVariance(mu=0., std=1.)  # Rescale time series
dataset = scaler.fit_transform(dataset)

# PAA transform (and inverse transform) of the data
```

(continues on next page)

(continued from previous page)

```
n_paa_segments = 10
paa = PiecewiseAggregateApproximation(n_segments=n_paa_segments)
paa_dataset_inv = paa.inverse_transform(paa.fit_transform(dataset))

# SAX transform
n_sax_symbols = 8
sax = SymbolicAggregateApproximation(n_segments=n_paa_segments,
                                     alphabet_size_avg=n_sax_symbols)
sax_dataset_inv = sax.inverse_transform(sax.fit_transform(dataset))

# 1d-SAX transform
n_sax_symbols_avg = 8
n_sax_symbols_slope = 8
one_d_sax = OneD_SymbolicAggregateApproximation(
    n_segments=n_paa_segments,
    alphabet_size_avg=n_sax_symbols_avg,
    alphabet_size_slope=n_sax_symbols_slope)
transformed_data = one_d_sax.fit_transform(dataset)
one_d_sax_dataset_inv = one_d_sax.inverse_transform(transformed_data)

plt.figure()
plt.subplot(2, 2, 1)  # First, raw time series
plt.plot(dataset[0].ravel(), "b-")
plt.title("Raw time series")

plt.subplot(2, 2, 2)  # Second, PAA
plt.plot(dataset[0].ravel(), "b-", alpha=0.4)
plt.plot(paa_dataset_inv[0].ravel(), "b-")
plt.title("PAA")

plt.subplot(2, 2, 3)  # Then SAX
plt.plot(dataset[0].ravel(), "b-", alpha=0.4)
plt.plot(sax_dataset_inv[0].ravel(), "b-")
plt.title("SAX, %d symbols" % n_sax_symbols)

plt.subplot(2, 2, 4)  # Finally, 1d-SAX
plt.plot(dataset[0].ravel(), "b-", alpha=0.4)
plt.plot(one_d_sax_dataset_inv[0].ravel(), "b-")
plt.title("1d-SAX, %d symbols"
          "(%dx%d)" % (n_sax_symbols_avg * n_sax_symbols_slope,
                       n_sax_symbols_avg,
                       n_sax_symbols_slope))

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.421 seconds)

4.5.3 Matrix Profile

This example presents a toy example of using Matrix Profile [1] for anomaly detection.

Matrix Profile transforms a time series into a sequence of 1-Nearest-Neighbor distances between its subseries.

[1] C. M. Yeh, Y. Zhu, L. Ulanova, N.Begum et al. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. ICDM 2016.

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms

from tslearn.matrix_profile import MatrixProfile

s_x = numpy.array(
    [-0.790, -0.765, -0.734, -0.700, -0.668, -0.639, -0.612, -0.587, -0.564,
     -0.544, -0.529, -0.518, -0.509, -0.502, -0.494, -0.488, -0.482, -0.475,
     -0.472, -0.470, -0.465, -0.464, -0.461, -0.458, -0.459, -0.460, -0.459,
     -0.458, -0.448, -0.431, -0.408, -0.375, -0.333, -0.277, -0.196, -0.090,
     0.047, 0.220, 0.426, 0.671, 0.962, 1.300, 1.683, 2.096, 2.510, 2.895,
     3.219, 3.463, 3.621, 3.700, 3.713, 3.677, 3.606, 3.510, 3.400, 3.280,
     3.158, 3.038, 2.919, 2.801, 2.676, 2.538, 2.382, 2.206, 2.016, 1.821,
     1.627, 1.439, 1.260, 1.085, 0.917, 0.758, 0.608, 0.476, 0.361, 0.259,
     0.173, 0.096, 0.027, -0.032, -0.087, -0.137, -0.179, -0.221, -0.260,
     -0.293, -0.328, -0.359, -0.385, -0.413, -0.437, -0.458, -0.480, -0.498,
     -0.512, -0.526, -0.536, -0.544, -0.552, -0.556, -0.561, -0.565, -0.568,
     -0.570, -0.570, -0.566, -0.560, -0.549, -0.532, -0.510, -0.480, -0.443,
     -0.402, -0.357, -0.308, -0.256, -0.200, -0.139, -0.073, -0.003, 0.066,
     0.131, 0.186, 0.229, 0.259, 0.276, 0.280, 0.272, 0.256, 0.234, 0.209,
     0.186, 0.162, 0.139, 0.112, 0.081, 0.046, 0.008, -0.032, -0.071, -0.110,
     -0.147, -0.180, -0.210, -0.235, -0.256, -0.275, -0.292, -0.307, -0.320,
     -0.332, -0.344, -0.355, -0.363, -0.367, -0.364, -0.351, -0.330, -0.299,
     -0.260, -0.217, -0.172, -0.128, -0.091, -0.060, -0.036, -0.022, -0.016,
     -0.020, -0.037, -0.065, -0.104, -0.151, -0.201, -0.253, -0.302, -0.347,
     -0.388, -0.426, -0.460, -0.491, -0.517, -0.539, -0.558, -0.575, -0.588,
     -0.600, -0.606, -0.607, -0.604, -0.598, -0.589, -0.577, -0.558, -0.531,
     -0.496, -0.454, -0.410, -0.364, -0.318, -0.276, -0.237, -0.203, -0.176,
     -0.157, -0.145, -0.142, -0.145, -0.154, -0.168, -0.185, -0.206, -0.230,
     -0.256, -0.286, -0.318, -0.351, -0.383, -0.414, -0.442, -0.467, -0.489,
     -0.508, -0.523, -0.535, -0.544, -0.552, -0.557, -0.560, -0.560, -0.557,
     -0.551, -0.542, -0.531, -0.519, -0.507, -0.494, -0.484, -0.476, -0.469,
     -0.463, -0.456, -0.449, -0.442, -0.435, -0.431, -0.429, -0.430, -0.435,
     -0.442, -0.452, -0.465, -0.479, -0.493, -0.506, -0.517, -0.526, -0.535,
     -0.548, -0.567, -0.592, -0.622, -0.655, -0.690, -0.728, -0.764, -0.795,
     -0.815, -0.823, -0.821]).reshape((-1, 1))

mp = MatrixProfile(subsequence_length=20, scale=False)
mp_series = mp.fit_transform([s_x])[0]
t_star = numpy.argmax(mp_series.ravel())

plt.figure()
ax = plt.subplot(2, 1, 1)  # First, raw time series
trans = mtransforms.blended_transform_factory(ax.transData, ax.transAxes)
plt.plot(s_x.ravel(), "b-")
plt.xlim([0, s_x.shape[0]])
plt.axvline(x=t_star,  c="red", linewidth=2)
plt.fill_between(x=[t_star, t_star+mp.subsequence_length], y1=0., y2=1.,
                 facecolor="r", alpha=.2, transform=trans)
plt.title("Raw time series")

plt.subplot(2, 1, 2)  # Second, Matrix Profile
```

(continues on next page)

(continued from previous page)

```
plt.plot(mp_series.ravel(), "b-")
plt.axvline(x=t_star,  c="red", linewidth=2, linestyle="dashed")
plt.xlim([0, s_x.shape[0]])
plt.title("Matrix Profile")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.160 seconds)

4.5.4 Distance and Matrix Profiles

This example illustrates how the matrix profile is calculated. For each segment of a timeseries with a specified length, the distances between each subsequence and that segment are calculated. The smallest distance is returned, except for trivial match on the location where the segment is extracted from which is equal to zero.

```
# Author: Gilles Vandewiele
# License: BSD 3 clause

import numpy
import matplotlib.patches as patches
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn.matrix_profile import MatrixProfile

import warnings
warnings.filterwarnings('ignore')

# Set a seed to ensure determinism
numpy.random.seed(42)

# Load the Trace dataset
X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Normalize the time series
scaler = TimeSeriesScalerMeanVariance()
X_train = scaler.fit_transform(X_train)

# Take the first time series
ts = X_train[0, :, :]

# We will take the spike as a segment
subseq_len = 20
start = 45
segment = ts[start:start + subseq_len]

# Create our matrix profile
matrix_profiler = MatrixProfile(subsequence_length=subseq_len, scale=True)
mp = matrix_profiler.fit_transform([ts]).flatten()
```

(continues on next page)

(continued from previous page)

```
# Create a grid for our plots
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)

# Plot our timeseries
ax1.plot(ts, c='b', label='time series')
ax1.add_patch(patches.Rectangle((start, numpy.min(ts) - 0.1), subseq_len,
                                numpy.max(ts) - numpy.min(ts) + 0.2,
                                facecolor='b', alpha=0.25,
                                label='segment'))
ax1.axvline(start, c='b', linestyle='--', lw=2, alpha=0.5,
            label='segment start')
ax1.legend(loc='lower right', ncol=4, fontsize=8,
           handletextpad=0.1, columnspacing=0.5)
ax1.set_title('The time series')

# Inset plot with our segment
fig_ax_in = ax1.inset_axes([0.5, 0.55, 0.2, 0.4])
fig_ax_in.plot(scaler.fit_transform(segment.reshape(1, -1, 1))[0], c='b')
ax1.indicate_inset(inset_ax=fig_ax_in, transform=ax1.transData,
                   bounds=[start, numpy.min(ts) - 0.1, subseq_len,
                           numpy.max(ts) - numpy.min(ts) + 0.2],
                   linestyle='--', alpha=0.75)
fig_ax_in.tick_params(labelleft=False, labelbottom=False)
fig_ax_in.xaxis.set_visible(False)
fig_ax_in.yaxis.set_visible(False)

# Calculate a distance profile, which represents the distance from each
# subsequence of the time series and the segment
distances = []
for i in range(len(ts) - subseq_len):
    scaled_ts = scaler.fit_transform(ts[i:i+subseq_len].reshape(1, -1, 1))
    scaled_segment = scaler.fit_transform(segment.reshape(1, -1, 1))
    distances.append(numpy.linalg.norm(scaled_ts - scaled_segment))

# Mask out the distances in the trivial match zone, get the nearest
# neighbor and put the old distances back in place so we can plot them.
distances = numpy.array(distances)
mask = list(range(start - subseq_len // 4, start + subseq_len // 4))
old_distances = distances[mask]
distances[mask] = numpy.inf
nearest_neighbor = numpy.argmin(distances)
dist_nn = distances[nearest_neighbor]
distances[mask] = old_distances

# Plot our distance profile
ax2.plot(distances, c='b')
ax2.set_title('Segment distance profile')
dist_diff = numpy.max(distances) - numpy.min(distances)
ax2.add_patch(patches.Rectangle((start - subseq_len // 4,
                                 numpy.min(distances) - 0.1),
                                subseq_len // 2,
                                dist_diff + 0.2,
                                facecolor='r', alpha=0.5,
                                label='exclusion zone'))
ax2.scatter(nearest_neighbor, dist_nn, c='r', marker='x', s=50,
            label='neighbor dist = {}'.format(numpy.around(dist_nn, 3)))
ax2.axvline(start, c='b', linestyle='--', lw=2, alpha=0.5,
```

(continues on next page)

(continued from previous page)

```
            label='segment start')
ax2.legend(loc='lower right', fontsize=8, ncol=3,
           handletextpad=0.1, columnspacing=0.5)

# Plot our matrix profile
ax3.plot(mp, c='b')
ax3.set_title('Matrix profile')
ax3.scatter(start, mp[start],
            c='r', marker='x', s=75,
            label='MP segment = {}'.format(numpy.around(mp[start], 3)))
ax3.axvline(start, c='b', linestyle='--', lw=2, alpha=0.5,
            label='segment start')
ax3.legend(loc='lower right', fontsize=8,
           handletextpad=0.1, columnspacing=0.25)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.691 seconds)

CHAPTER 5

Citing *tslearn*

If you use *tslearn* in a scientific publication, we would appreciate citations:

Bibtex entry:

```
@article{JMLR:v21:20-091,
  author  = {Romain Tavenard and Johann Faouzi and Gilles Vandewiele and
             Felix Divo and Guillaume Androz and Chester Holtz and
             Marie Payne and Roman Yurchak and Marc Ru{\ss}wurm and
             Kushal Kolar and Eli Woods},
  title   = {Tslearn, A Machine Learning Toolkit for Time Series Data},
  journal = {Journal of Machine Learning Research},
  year    = {2020},
  volume  = {21},
  number  = {118},
  pages   = {1-6},
  url     = {http://jmlr.org/papers/v21/20-091.html}
}
```

Bibliography

[1] F. Petitjean, A. Ketterlin & P. Gancarski. A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition, Elsevier, 2011, Vol. 44, Num. 3, pp. 678-693

[2] D. Schultz and B. Jain. Nonsmooth Analysis and Subgradient Methods for Averaging in Dynamic Time Warping Spaces. Pattern Recognition, 74, 340-358.

[1] F. Petitjean, A. Ketterlin & P. Gancarski. A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition, Elsevier, 2011, Vol. 44, Num. 3, pp. 678-693

[2] D. Schultz and B. Jain. Nonsmooth Analysis and Subgradient Methods for Averaging in Dynamic Time Warping Spaces. Pattern Recognition, 74, 340-358.

[1] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.

[1] Kernel k-means, Spectral Clustering and Normalized Cuts. Inderjit S. Dhillon, Yuqiang Guan, Brian Kulis. KDD 2004.

[2] Fast Global Alignment Kernels. Marco Cuturi. ICML 2011.

[1] J. Paparrizos & L. Gravano. k-Shape: Efficient and Accurate Clustering of Time Series. SIGMOD 2015. pp. 1855-1870.

[1] Peter J. Rousseeuw (1987). "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis". Computational and Applied Mathematics 20: 53-65.

[2] Wikipedia entry on the Silhouette Coefficient

[1] A. Bagnall, J. Lines, W. Vickers and E. Keogh, The UEA & UCR Time Series Classification Repository, www.timeseriesclassification.com

[1] A. Bagnall, J. Lines, W. Vickers and E. Keogh, The UEA & UCR Time Series Classification Repository, www.timeseriesclassification.com

[1] A. Dachraoui, A. Bondu & A. Cornuejols. Early classification of time series as a non myopic sequential decision making problem. ECML/PKDD 2015

[1] C. M. Yeh, Y. Zhu, L. Ulanova, N.Begum et al. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. ICDM 2016.

[2] STUMPY documentation https://stumpy.readthedocs.io/en/latest/

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[1] M. Cuturi, "Fast global alignment kernels," ICML 2011.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[2] Z. Zhang, R. Tavenard, A. Bailly, X. Tang, P. Tang, T. Corpetti Dynamic time warping under limited warping path length. Information Sciences, vol. 393, pp. 91–107, 2017.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[2] Z. Zhang, R. Tavenard, A. Bailly, X. Tang, P. Tang, T. Corpetti Dynamic time warping under limited warping path length. Information Sciences, vol. 393, pp. 91–107, 2017.

[1] M. Cuturi, "Fast global alignment kernels," ICML 2011.

[1] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.

[1] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.

[1] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.

[1] Keogh, E. Exact indexing of dynamic time warping. In International Conference on Very Large Data Bases, 2002. pp 406-417.

[1] Keogh, E. Exact indexing of dynamic time warping. In International Conference on Very Large Data Bases, 2002. pp 406-417.

[1] M. Cuturi, "Fast global alignment kernels," ICML 2011.

[1] M. Cuturi, "Fast global alignment kernels," ICML 2011.

[1] S. Malinowski, T. Guyet, R. Quiniou, R. Tavenard. 1d-SAX: a Novel Symbolic Representation for Time Series. IDA 2013.

[1] S. Malinowski, T. Guyet, R. Quiniou, R. Tavenard. 1d-SAX: a Novel Symbolic Representation for Time Series. IDA 2013.

[1] S. Malinowski, T. Guyet, R. Quiniou, R. Tavenard. 1d-SAX: a Novel Symbolic Representation for Time Series. IDA 2013.

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[1] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[1] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

[1] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

Python Module Index

t

Index

G

I

K

L

M

N

O

P

R

S

T

U