tslearn Documentation

Release 0.2.5

Romain Tavenard

Oct 11, 2019

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.

If you plan to use the `shapelets` module from `tslearn`, `keras` and `tensorflow` should also be installed.

CHAPTER 1

Installation

1.1 Using conda

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

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

1.2 Using PyPI

Using `pip` should also work fine:

```
pip install tslearn
```

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

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:

```
pip install git+https://github.com/rtavenar/tslearn.git
```

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

CHAPTER 2

Navigation

From here, you can navigate to:

2.1 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.

2.1.1 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` module:

```
>>> 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])
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

2.1.2 Importing standard time series datasets

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

```
>>> 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* module.

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 sparated 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* module:

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

2.1.3 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.

2.2 Methods for variable-length time series datasets

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]
```

2.2.1 Classification

- *KNeighborsTimeSeriesClassifier*
- *TimeSeriesSVC*

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)
```

2.2.2 Regression

- *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)
```

2.2.3 Clustering

- *GlobalAlignmentKernelKMeans*
- *TimeSeriesKMeans*
- *silhouette_score*

Examples

```
from tslearn.clustering import GlobalAlignmentKernelKMeans
gak_km = GlobalAlignmentKernelKMeans(n_clusters=2)
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")
```

2.2.4 Barycenter computation

- *dtw_barycenter_averaging*
- *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)
```

2.2.5 Model selection

Also, model selection tools offered by *sklearn* 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)
```

2.2.6 Resampling

- *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)
```

2.3 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.generators	The *tslearn.generators* module gathers synthetic time series dataset generation routines.
tslearn.metrics	The *tslearn.metrics* module gathers time series similarity metrics.
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.
tslearn.shapelets	The *tslearn.shapelets* module gathers Shapelet-based algorithms.
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.

2.3.1 tslearn.barycenters

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

Functions

euclidean_barycenter(X[, weights])	Standard Euclidean barycenter computed from a set of time series.
dtw_barycenter_averaging(X[, ...])	DTW Barycenter Averaging (DBA) method.

Continued on next page

Table 2 – continued from previous page

softdtw_barycenter(X[, gamma, weights, ...])	Compute barycenter (time series averaging) under the soft-DTW geometry.

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*

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*, *verbose=False*)

DTW Barycenter Averaging (DBA) method.

DBA was originally presented in [1].

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.

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]

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, barycenter_size=3)
array([[1.5       ],
       [3.        ],
       [4.33333333]])
```

Examples using `tslearn.barycenters.dtw_barycenter_averaging`

- *Barycenters*

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 geometry.

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.

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*

2.3.2 tslearn.clustering

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

Classes

GlobalAlignmentKernelKMeans([n_clusters, ...])	Global Alignment 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.

tslearn.clustering.GlobalAlignmentKernelKMeans

class tslearn.clustering.**GlobalAlignmentKernelKMeans**(*n_clusters=3*, *max_iter=50*, *tol=1e-06*, *n_init=1*, *sigma=1.0*, *n_jobs=None*, *verbose=False*, *random_state=None*)

Global Alignment Kernel K-means.

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.

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*)

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 [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.

References

Kernel k-means, Spectral Clustering and Normalized Cuts. Inderjit S. Dhillon, Yuqiang Guan, Brian Kulis. KDD 2004.

Fast Global Alignment Kernels. Marco Cuturi. ICML 2011.

Examples

```
>>> from tslearn.generators import random_walks
>>> X = random_walks(n_ts=50, sz=32, d=1)
>>> gak_km = GlobalAlignmentKernelKMeans(n_clusters=3,
...                                      random_state=0).fit(X)
```

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.

Methods

fit(self, X[, y, sample_weight])	Compute kernel k-means clustering.
fit_predict(self, X[, y])	Fit kernel k-means clustering using X and then predict the closest cluster each time series in X belongs to.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the closest cluster each time series in X belongs to.
set_params(self, **params)	Set the parameters of this estimator.

fit (*self*, *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 (*self*, *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.

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

Parameters

deep [boolean, optional] 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(*self*, *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(*self*, ***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.

Returns

self

Examples using `tslearn.clustering.GlobalAlignmentKernelKMeans`

- *Kernel k-means*

`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 [R8d0e3400ad02-1].

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.

Notes

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

References

[R8d0e3400ad02-1]

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)
```

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.

Methods

fit(self, X[, y])	Compute k-Shape clustering.
fit_predict(self, X[, y])	Fit k-Shape clustering using X and then predict the closest cluster each time series in X belongs to.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the closest cluster each time series in X belongs to.
set_params(self, **params)	Set the parameters of this estimator.

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

Parameters

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

y Ignored

fit_predict (*self*, *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.

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

Parameters

deep [boolean, optional] 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(*self*, *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(*self*, ***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.

Returns

self

Examples using `tslearn.clustering.KShape`

- *KShape*

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=False*, *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. Value associated to the *"gamma_sdtw"* key corresponds to the gamma parameter in Soft-DTW.

Deprecated since version 0.2: *"gamma_sdtw"* as a key for *metric_params* is deprecated in version 0.2 and will be removed in 0.4.

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 [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 [{'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.

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, 3, 1)
```

Attributes

labels_ [numpy.ndarray] Labels of each point.

cluster_centers_ [numpy.ndarray] Cluster centers.

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

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

Methods

fit(self, X[, y])	Compute k-means clustering.
fit_predict(self, X[, y])	Fit k-means clustering using X and then predict the closest cluster each time series in X belongs to.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the closest cluster each time series in X belongs to.
set_params(self, **params)	Set the parameters of this estimator.

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

Compute k-means clustering.

Parameters

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

y Ignored

fit_predict (*self*, *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.

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

Parameters

deep [boolean, optional] 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(*self*, *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(*self*, ***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.

Returns

self

Examples using `tslearn.clustering.TimeSeriesKMeans`

- *k-means*

Functions

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

tslearn.clustering.silhouette_score

tslearn.clustering.**silhouette_score**(*X*, *labels*, *metric=None*, *sample_size=None*, *metric_params=None*, *n_jobs=None*, *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. Value associated to the *"gamma_sdtw"* key corresponds to the gamma parameter in Soft-DTW.

Deprecated since version 0.2: *"gamma_sdtw"* as a key for *metric_params* is deprecated in version 0.2 and will be removed in 0.4.

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.

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...
```

(continues on next page)

(continued from previous page)

```
>>> 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...
```

2.3.3 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.

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 [R25f7dec5cd6c-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

[R25f7dec5cd6c-1]

Methods

baseline_accuracy(self[, list_datasets, ...])	Report baseline performances as provided by UEA/UCR website.
cache_all(self)	Cache all datasets from the UCR/UEA archive for later use.
list_cached_datasets(self)	List datasets from the UCR/UEA archive that are available in cache.
list_datasets(self)	List datasets in the UCR/UEA archive.

Continued on next page

Table 9 – continued from previous page

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

baseline_accuracy (*self*, *list_datasets=None*, *list_methods=None*)
Report baseline performances as provided by UEA/UCR website.

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 (*self*)
Cache all datasets from the UCR/UEA archive for later use.

list_cached_datasets (*self*)
List datasets from the UCR/UEA archive that are available in cache.

Examples

```
>>> l = UCR_UEA_datasets().list_cached_datasets()
>>> 0 <= len(l) <= len(UCR_UEA_datasets().list_datasets())
True
```

list_datasets (*self*)
List datasets in the UCR/UEA archive.

Examples

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

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

tslearn.datasets.CachedDatasets

class tslearn.datasets.**CachedDatasets**
A convenience class to access cached time series datasets.

When using the Trace dataset, please cite [R06dcc6755300-1].

References

[R06dcc6755300-1]

Methods

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

list_datasets (*self*)
List cached datasets.

load_dataset (*self*, *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`

- *SVM and GAK*
- *k-NN search*
- *KShape*
- *Barycenters*
- *Kernel k-means*
- *Learning Shapelets*
- *Soft-DTW weighted barycenters*
- *Learning Shapelets*
- *k-means*
- *Hyper-parameter tuning of a Pipeline with KNeighborsTimeSeriesClassifier*

2.3.4 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.

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*

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`

- *DTW computation*
- *LB_Keogh*
- *PAA and SAX features*

2.3.5 tslearn.metrics

This modules delivers time-series specific metrics to be used at the core of machine learning algorithms.

Dynamic Time Warping

Dynamic Time Warping (DTW) is one of the most popular time-series dissimilarity scores. It consists in computing Euclidean distance between aligned time series. To do so, it needs to both determine the optimal alignment between time series and compute the associated cost, which is done (using dynamic programming) by computing the optimal path in a similarity matrix.

In the implementation included in `tslearn`, standard transition steps are used: diagonal, horizontal and vertical and DTW is computed as the Euclidean distance along the optimal path.

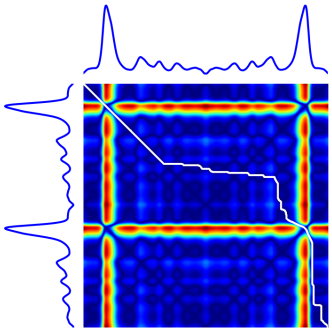


Fig. 1: Example DTW path.

The *tslearn.metrics* module gathers time series similarity metrics.

Functions

cdist_dtw(dataset1[, dataset2, ...])	Compute cross-similarity matrix using Dynamic Time Warping (DTW) similarity measure.
cdist_gak(dataset1[, dataset2, sigma, n_jobs])	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_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.
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.

Continued on next page

Table 12 – continued from previous page

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.

tslearn.metrics.cdist_dtw

tslearn.metrics.**cdist_dtw**(*dataset1*, *dataset2=None*, *global_constraint=None*, *sakoe_chiba_radius=1*, *itakura_max_slope=2.0*, *n_jobs=None*)

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 P is the alignment path:

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

DTW was originally presented in [1].

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 (default: 1)] Radius to be used for Sakoe-Chiba band global constraint. Used only if `global_constraint="sakoe_chiba"`.

itakura_max_slope [float (default: 2.)] Maximum slope for the Itakura parallelogram constraint. Used only if `global_constraint="itakura"`.

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.

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]])
```

tslearn.metrics.cdist_gak

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

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.

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]])
```

tslearn.metrics.dtw

tslearn.metrics.**dtw**(*s1*, *s2*, *global_constraint=None*, *sakoe_chiba_radius=1*, *itakura_max_slope=2.0*)

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 P is the optimal alignment path:

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

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].

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 (default: 1)] Radius to be used for Sakoe-Chiba band global constraint. Used only if `global_constraint="sakoe_chiba"`.

itakura_max_slope [float (default: 2.)] Maximum slope for the Itakura parallelogram constraint. Used only if `global_constraint="itakura"`.

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
```

tslearn.metrics.dtw_path

`tslearn.metrics.`**`dtw_path`**(*s1*, *s2*, *global_constraint=None*, *sakoe_chiba_radius=1*, *itakura_max_slope=2.0*)

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 P is the alignment path:

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

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].

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 (default: 1)] Radius to be used for Sakoe-Chiba band global constraint. Used only if `global_constraint="sakoe_chiba"`.

itakura_max_slope [float (default: 2.)] Maximum slope for the Itakura parallelogram constraint. Used only if `global_constraint="itakura"`.

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

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*

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 P is the alignment path:

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

Compared to traditional DTW, here, border constraints on admissible paths P are relaxed such that $P_0 = (0, ?)$ and $P_L = (N - 1, ?)$ 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 A query time series.

longseq 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

Examples

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

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...
```

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].

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...
```

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].

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.        ]])
```

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]. This normalized version is defined as: *sdtw_(x,y) := sdtw(x,y) - 1/2(sdtw(x,x)+sdtw(y,y))* and ensures that all returned values are positive and that *sdtw_(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
```

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.],
```

(continues on next page)

(continued from previous page)

```
       [2.],
       [1.],
       [1.]])
>>> env_up
array([[2.],
       [3.],
       [3.],
       [3.],
       [2.]])
```

Examples using `tslearn.metrics.lb_envelope`

- *LB_Keogh*

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*

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 ($sigma$) 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...
```

Examples using `tslearn.metrics.sigma_gak`

- *Kernel k-means*

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
$gamma$ 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...
```

2.3.6 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.

tslearn.neighbors.KNeighborsTimeSeries

class tslearn.neighbors.**KNeighborsTimeSeries**(*n_neighbors=5*, *metric='dtw'*, *metric_params=None*, *n_jobs=None*)

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'}]

(default: 'dtw') Metric to be used at the core of the nearest neighbor procedure. DTW is described in more details in *tslearn.metrics*. Other metrics are described in scipy.spatial.distance doc.

metric_params [dict or None (default: None)] Dictionnary of metric parameters. 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.

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.]])
>>> ind
array([[0]])
>>> knn2 = KNeighborsTimeSeries(n_neighbors=10,
...                             metric="euclidean").fit(time_series)
>>> knn2.kneighbors(return_distance=False)
array([[2, 1],
       [2, 0],
       [0, 1]])
```

Methods

fit(self, X[, y])	Fit the model using X as training data
get_params(self[, deep])	Get parameters for this estimator.
kneighbors(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
kneighbors_graph(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
radius_neighbors(self[, X, radius, ...])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph(self[, X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
set_params(self, **params)	Set the parameters of this estimator.

fit (*self*, *X*, *y=None*)
Fit the model using X as training data

Parameters

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

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

Parameters

deep [boolean, optional] 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 (*self*, *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 (*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_query, n_features), or (n_query, 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 matrix in CSR format, shape = [n_samples, 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) # doctest: +ELLIPSIS
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

radius_neighbors(*self*, *X=None*, *radius=None*, *return_distance=True*)

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

Returns

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.

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) # doctest: +ELLIPSIS
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0])) # doctest: +ELLIPSIS
[1.5 0.5]
>>> print(np.asarray(rng[1][0])) # doctest: +ELLIPSIS
[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 (*self*, *X=None*, *radius=None*, *mode='connectivity'*)

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, shape = [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] 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.

Returns

A [sparse matrix in CSR format, shape = [n_samples, n_samples]] 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)
```

(continues on next page)

(continued from previous page)

```
>>> neigh.fit(X) # doctest: +ELLIPSIS
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

set_params(*self*, ***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.

Returns

self

Examples using `tslearn.neighbors.KNeighborsTimeSeries`

- *k-NN search*
- *Nearest neighbors*

tslearn.neighbors.KNeighborsTimeSeriesClassifier

class tslearn.neighbors.**KNeighborsTimeSeriesClassifier**(*n_neighbors=5*, *weights='uniform'*, *metric='dtw'*, *metric_params=None*, *n_jobs=None*)

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* 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.

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])
```

Methods

fit(self, X, y)	Fit the model using X as training data and y as target values
get_params(self[, deep])	Get parameters for this estimator.
kneighbors(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
kneighbors_graph(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(self, X)	Predict the class labels for the provided data
predict_proba(self, X)	Predict the class probabilities for the provided data
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

fit (*self*, *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.

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

Parameters

deep [boolean, optional] 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 (*self*, *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 (*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_query, n_features), or (n_query, 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 matrix in CSR format, shape = [n_samples, 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) # doctest: +ELLIPSIS
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

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

Parameters

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

predict_proba (*self*, *X*)
Predict the class probabilities for the provided data

Parameters

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

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `tslearn.neighbors.KNeighborsTimeSeriesClassifier`

- *Hyper-parameter tuning of a Pipeline with KNeighborsTimeSeriesClassifier*
- *Nearest neighbors*

2.3.7 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.

tslearn.piecewise.OneD_SymbolicAggregateApproximation

class tslearn.piecewise.**OneD_SymbolicAggregateApproximation**(*n_segments*, *alphabet_size_avg*, *alphabet_size_slope*, *sigma_l=None*)

One-D Symbolic Aggregate approXimation (1d-SAX) transformation.

1d-SAX was originally presented in [Rbedf9fcb0133-1].

Parameters

n_segments [int] Number of PAA segments to compute.

alphabet_size_avg [int] Number of SAX symbols to use to describe average values.

alphabet_size_slope [int] 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 [Rbedf9fcb0133-1] is used: $\sigma_L = \sqrt{0.03/L}$ where L is the length of each segment.

Notes

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

References

[Rbedf9fcb0133-1]

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],
```

(continues on next page)

(continued from previous page)

```
        [ 0.33724488]],
<BLANKLINE>
       [[ 0.33724488],
        [ 1.01173463],
        [-0.33724488],
        [-1.01173463],
        [ 1.01173463],
        [ 0.33724488]]])
>>> one_d_sax.fit(data).sigma_l
1.0
```

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.

Methods

distance(self, ts1, ts2)	Compute distance between 1d-SAX representations as defined in [1].
distance_1d_sax(self, sax1, sax2)	Compute distance between 1d-SAX representations as defined in [1].
distance_paa(self, paa1, paa2)	Compute distance between PAA representations as defined in [1].
distance_sax(self, sax1, sax2)	Compute distance between SAX representations as defined in [1].
fit(self, X[, y])	Fit a 1d-SAX representation.
fit_transform(self, X[, y])	Fit a 1d-SAX representation and transform the data accordingly.
inverse_transform(self, X)	Compute time series corresponding to given 1d-SAX representations.
transform(self, X[, y])	Transform a dataset of time series into its 1d-SAX representation.

distance(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *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

inverse_transform(*self*, *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.

transform(*self*, *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*

tslearn.piecewise.PiecewiseAggregateApproximation

class tslearn.piecewise.**PiecewiseAggregateApproximation**(*n_segments*)
Piecewise Aggregate Approximation (PAA) transformation.

PAA was originally presented in [R48c3dfce0d76-1].

Parameters

n_segments [int] Number of PAA segments to compute

Notes

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

References

[R48c3dfce0d76-1]

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(self, ts1, ts2)	Compute distance between PAA representations as defined in [1].
distance_paa(self, paa1, paa2)	Compute distance between PAA representations as defined in [1].
fit(self, X[, y])	Fit a PAA representation.
fit_transform(self, X[, y])	Fit a PAA representation and transform the data accordingly.
inverse_transform(self, X)	Compute time series corresponding to given PAA representations.
transform(self, X[, y])	Transform a dataset of time series into its PAA representation.

distance(*self*, *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(*self*, *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(*self*, *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(*self*, *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

inverse_transform(*self*, *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.

transform(*self*, *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*

tslearn.piecewise.SymbolicAggregateApproximation

class tslearn.piecewise.**SymbolicAggregateApproximation**(*n_segments*, *alphabet_size_avg*)
Symbolic Aggregate approXimation (SAX) transformation.

SAX was originally presented in [R83f60d91f77e-1].

Parameters

n_segments [int] Number of PAA segments to compute

alphabet_size_avg [int] Number of SAX symbols to use

Notes

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

References

[R83f60d91f77e-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]]])
```

(continues on next page)

(continued from previous page)

```
>>> 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]]])
```

Attributes

breakpoints_avg_ [numpy.ndarray of shape (alphabet_size - 1,)] List of breakpoints used to generate SAX symbols

Methods

distance(self, ts1, ts2)	Compute distance between SAX representations as defined in [1].
distance_paa(self, paa1, paa2)	Compute distance between PAA representations as defined in [1].
distance_sax(self, sax1, sax2)	Compute distance between SAX representations as defined in [1].
fit(self, X[, y])	Fit a SAX representation.
fit_transform(self, X[, y])	Fit a SAX representation and transform the data accordingly.
inverse_transform(self, X)	Compute time series corresponding to given SAX representations.
transform(self, X[, y])	Transform a dataset of time series into its SAX representation.

distance(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *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

inverse_transform(*self*, *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.

transform(*self*, *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*
- *Nearest neighbors*

2.3.8 tslearn.preprocessing

The *tslearn.preprocessing* module gathers time series scalers.

Classes

TimeSeriesScalerMeanVariance([mu, std])	Scaler for time series.
TimeSeriesScalerMinMax([value_range, min, max])	Scaler for time series.
TimeSeriesResampler(sz)	Resampler for time series.

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.

Examples

```
>>> TimeSeriesScalerMeanVariance(mu=0.,
...                              std=1.).fit_transform([[0, 3, 6]])
array([[[-1.22474487],
        [ 0.        ],
        [ 1.22474487]]])
```

Methods

fit(self, X[, y])	A dummy method such that it complies to the sklearn requirements.
fit_transform(self, X[, y])	Fit to data, then transform it.
transform(self, X, **kwargs)	Fit to data, then transform it.

fit (*self*, *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 (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

transform (*self*, *X*, ***kwargs*)
Fit to data, then transform it.

Parameters

X Time series dataset to be rescaled

Returns

numpy.ndarray Rescaled time series dataset

Examples using `tslearn.preprocessing.TimeSeriesScalerMeanVariance`

- *DTW computation*
- *LB_Keogh*
- *KShape*

- *Kernel k-means*
- *PAA and SAX features*
- *k-means*

tslearn.preprocessing.TimeSeriesScalerMinMax

class tslearn.preprocessing.**TimeSeriesScalerMinMax**(*value_range=(0.0, 1.0)*, *min=None*, *max=None*)

Scaler for time series. Scales time series so that their span in each dimension is between `min` and `max`.

Parameters

value_range [tuple (default: (0., 1.))] The minimum and maximum value for the output time series.

min [float (default: 0.)] Minimum value for output time series.

Deprecated since version 0.2: min is deprecated in version 0.2 and will be removed in 0.4. Use value_range instead.

max [float (default: 1.)] Maximum value for output time series.

Deprecated since version 0.2: min is deprecated in version 0.2 and will be removed in 0.4. Use value_range instead.

Notes

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

Examples

```
>>> TimeSeriesScalerMinMax(value_range=(1., 2.)).fit_transform([[0, 3, 6]])
array([[[1. ],
        [1.5],
        [2. ]]])
```

Methods

fit(self, X[, y])	A dummy method such that it complies to the sklearn requirements.
fit_transform(self, X[, y])	Fit to data, then transform it.
transform(self, X[, y])	Will normalize (min-max) each of the timeseries.

fit(*self*, *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(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

transform(*self*, *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] Time series dataset to be rescaled.

Returns

numpy.ndarray Rescaled time series dataset.

Examples using `tslearn.preprocessing.TimeSeriesScalerMinMax`

- *SVM and GAK*
- *Learning Shapelets*
- *Soft-DTW weighted barycenters*
- *Learning Shapelets*
- *Hyper-parameter tuning of a Pipeline with KNeighborsTimeSeriesClassifier*
- *Nearest neighbors*

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(self, X[, y])	A dummy method such that it complies to the sklearn requirements.
fit_transform(self, X[, y])	Fit to data, then transform it.
transform(self, X, **kwargs)	Fit to data, then transform it.

fit (*self*, *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 (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

transform (*self*, *X*, ***kwargs*)
Fit to data, then transform it.

Parameters

X [array-like] Time series dataset to be resampled.

Returns

numpy.ndarray Resampled time series dataset.

Examples using `tslearn.preprocessing.TimeSeriesResampler`

- *k-means*

2.3.9 tslearn.shapelets

The *tslearn.shapelets* module gathers Shapelet-based algorithms.

It depends on the *keras* library for optimization.

Functions

grabocka_params_to_shapelet_size_dict(n_ts, ...)	Compute number and length of shapelets.

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 Dictionnary 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*
- *Learning Shapelets*

Classes

ShapeletModel([n_shapelets_per_size, ...])	Learning Time-Series Shapelets model.
SerializableShapeletModel([...])	Serializable variant of the Learning Time-Series Shapelets model.

tslearn.shapelets.ShapeletModel

class tslearn.shapelets.**ShapeletModel**(*n_shapelets_per_size=None*, *max_iter=100*, *batch_size=256*, *verbose=0*, *verbose_level=None*, *optimizer='sgd'*, *weight_regularizer=0.0*, *shapelet_length=0.15*, *total_lengths=3*, *random_state=None*)

Learning Time-Series Shapelets model.

Learning Time-Series Shapelets was originally presented in [R1290393aa4b0-1].

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)

max_iter: int (default: 100) Number of training epochs.

batch_size: int (default: 256) Batch size to be used.

verbose_level: {0, 1, 2} (default: 0) *keras* verbose level.

Deprecated since version 0.2: verbose_level is deprecated in version 0.2 and will be removed in 0.4. Use *verbose* instead.

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 ts length

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]

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*.

Notes

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

References

[R1290393aa4b0-1]

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 = ShapeletModel(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)
```

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.

Methods

fit(self, X, y)	Learn time-series shapelets.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_weights(self[, layer_name])	Return model weights (or weights for a given layer if *layer_name* is provided).
locate(self, X)	Compute shapelet match location for a set of time series.
predict(self, X)	Predict class for a given set of time series.
predict_proba(self, X)	Predict class probability for a given set of time series.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Generate shapelet transform for a set of time series.

fit (*self*, *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(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

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

Parameters

deep [boolean, optional] 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(*self*, *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 = ShapeletModel(n_shapelets_per_size={10: 5}, max_iter=0,
...                     verbose=0)
>>> clf.fit(X, y).get_weights("classification")[0].shape
(5, 3)
```

locate(*self*, *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.

predict (*self*, *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 (*self*, *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 (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

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 = ShapeletModel(n_shapelets_per_size={3: 2, 4: 1},
...                       max_iter=1)
>>> _ = model.fit(X, y)
```

(continues on next page)

(continued from previous page)

```
>>> model.shapelets_as_time_series_.shape
(3, 4, 1)
```

transform(*self*, *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.ShapeletModel`

- *Learning Shapelets*
- *Learning Shapelets*

tslearn.shapelets.SerializableShapeletModel

class tslearn.shapelets.**SerializableShapeletModel**(*n_shapelets_per_size=None*, *max_iter=100*, *batch_size=256*, *verbose=0*, *verbose_level=None*, *learning_rate=0.01*, *weight_regularizer=0.0*, *shapelet_length=0.3*, *total_lengths=3*, *random_state=None*)

Serializable variant of the Learning Time-Series Shapelets model.

Learning Time-Series Shapelets was originally presented in [R9a1317388c84-1].

Parameters

n_shapelets_per_size: dict (default: None) Dictionary giving, for each shapelet size (key), the number of such shapelets to be trained (value)

max_iter: int (default: 100) Number of training epochs.

batch_size: int (default:256) Batch size to be used.

verbose_level: {0, 1, 2} (default: 0) *keras* verbose level.

Deprecated since version 0.1: min is deprecated in version 0.1 and will be removed in 0.2.

verbose: {0, 1, 2} (default: 0) *keras* verbose level.

learning_rate: float (default: 0.01) Learning rate to be used for the SGD optimizer.

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 ts length

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]

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*.

Notes

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

References

[R9a1317388c84-1]

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 = SerializableShapeletModel(n_shapelets_per_size={4: 5},
...                                 max_iter=1, verbose=0,
...                                 learning_rate=0.01)
>>> _ = clf.fit(X, y)
>>> clf.shapelets_.shape[0]
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)
```

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.

Methods

fit(self, X, y)	Learn time-series shapelets.

Continued on next page

Table 27 – continued from previous page

fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_weights(self[, layer_name])	Return model weights (or weights for a given layer if *layer_name* is provided).
locate(self, X)	Compute shapelet match location for a set of time series.
predict(self, X)	Predict class for a given set of time series.
predict_proba(self, X)	Predict class probability for a given set of time series.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Generate shapelet transform for a set of time series.

fit (*self*, *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 (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

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

Parameters

deep [boolean, optional] 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 (*self*, *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 = ShapeletModel(n_shapelets_per_size={10: 5}, max_iter=0,
...                     verbose=0)
>>> clf.fit(X, y).get_weights("classification")[0].shape
(5, 3)
```

locate (*self*, *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.

predict (*self*, *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 (*self*, *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 (*self*, *X*, *y*, *sample_weight=None*)

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

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 = ShapeletModel(n_shapelets_per_size={3: 2, 4: 1},
...                       max_iter=1)
>>> _ = model.fit(X, y)
>>> model.shapelets_as_time_series_.shape
(3, 4, 1)
```

transform(*self*, *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.

2.3.10 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.

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=False*, *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=True)] Whether to enable probability estimates. This must be enabled prior to calling *fit*, and will slow down that method.

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 [bool, default: False] 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*.

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_time_series_(X)
>>> 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)
```

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.

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

Methods

get_params(self[, deep])	Get parameters for this estimator.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

decision_function
fit
predict
predict_log_proba
predict_proba
support_vectors_time_series_

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

Parameters

deep [boolean, optional] 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.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `tslearn.svm.TimeSeriesSVC`

- *SVM and GAK*

`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=False*, *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 [bool, default: False] 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.

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_time_series_(X)
>>> sv.shape  # doctest: +ELLIPSIS
(..., 64, 2)
>>> sv.shape[0] <= 20
True
```

Attributes

support_ [array-like, shape = [n_SV]] Indices of support vectors.

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

Methods

get_params(self[, deep])	Get parameters for this estimator.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

fit	
predict	
support_vectors_time_series_	

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

Parameters

deep [boolean, optional] 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.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

2.3.11 tslearn.utils

The *tslearn.utils* module includes various utilities.

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.
ts_size(ts)	Returns actual time series size.
ts_zeros(sz[, d])	Returns a time series made of zero values.
load_timeseries_txt(fname)	Loads a time series dataset from disk.
save_timeseries_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, 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.

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.]])
```

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.]]])
```

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.]])
```

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],
```

(continues on next page)

(continued from previous page)

```
...           [numpy.nan, numpy.nan]])
4
```

Examples using `tslearn.utils.ts_size`

- *Learning Shapelets*

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.]])
>>> ts_zeros(5).shape
(5, 1)
```

tslearn.utils.load_timeseries_txt

`tslearn.utils.`**`load_timeseries_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_timeseries_txt` Save time series to disk

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4], [1, 2, 3]])
>>> save_timeseries_txt("tmp-tslearn-test.txt", dataset)
>>> reloaded_dataset = load_timeseries_txt("tmp-tslearn-test.txt")
```

tslearn.utils.save_timeseries_txt

tslearn.utils.**save_timeseries_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_timeseries_txt Load time series from disk

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4], [1, 2, 3]])
>>> save_timeseries_txt("tmp-tslearn-test.txt", dataset)
```

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
```

tslearn.utils.check_dims

tslearn.utils.**check_dims**(*X*, *X_fit=None*, *extend=True*)

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 dimensions, except the first one, of X_fit and X match.

Parameters

X [array-like] The first array to be compared.

X_fit [array-like or None (default: None)] The second array to be compared, which is created during fit. If None, then only perform reshaping of X, if necessary.

extend [boolean (default: True)] Whether to reshape X, if it is 2-dimensional.

Returns

array Reshaped X array

Raises

ValueError Will raise exception if X is None or (if X_fit is provided) one of the dimensions, except the first, does not match.

Examples

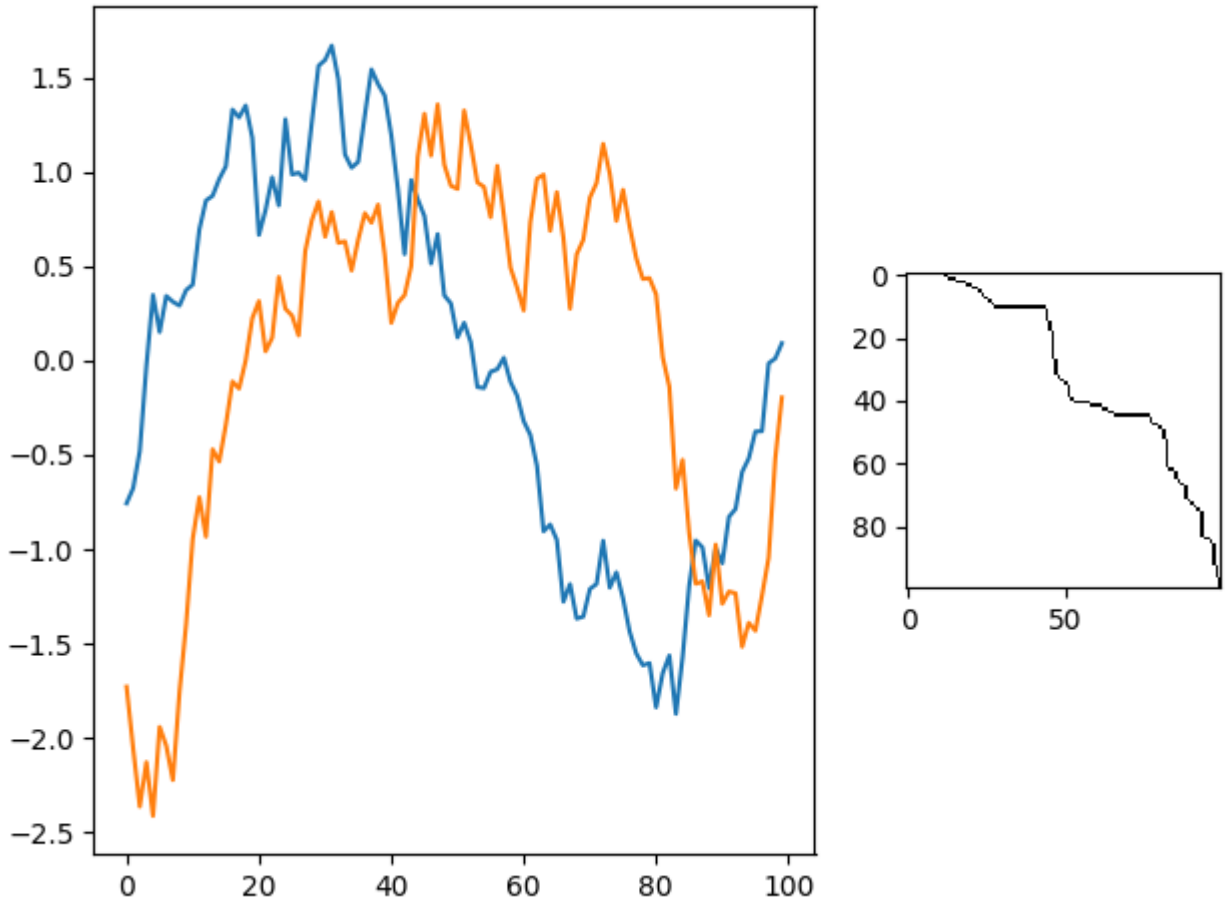
```
>>> X = numpy.empty((10, 3))
>>> check_dims(X).shape
(10, 3, 1)
>>> X = numpy.empty((10, 3, 1))
>>> check_dims(X).shape
(10, 3, 1)
>>> X_fit = numpy.empty((5, 3, 1))
>>> check_dims(X, X_fit).shape
(10, 3, 1)
>>> X_fit = numpy.empty((5, 3, 2))
>>> check_dims(X, X_fit)  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
ValueError: Dimensions (except first) must match! ((5, 3, 2) and (10, 3, 1)
are passed shapes)
```

2.4 Gallery of examples

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.1 DTW computation

This example illustrates DTW computation between time series and plots the optimal alignment path.




```
# 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 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)

path, sim = metrics.dtw_path(dataset_scaled[0], dataset_scaled[1])

matrix_path = numpy.zeros((sz, sz), dtype=numpy.int)
for i, j in path:
    matrix_path[i, j] = 1

plt.figure()
```

(continues on next page)

(continued from previous page)

```
plt.subplot2grid((1, 3), (0, 0), colspan=2)
plt.plot(numpy.arange(sz), dataset_scaled[0, :, 0])
plt.plot(numpy.arange(sz), dataset_scaled[1, :, 0])
plt.subplot(1, 3, 3)
plt.imshow(matrix_path, cmap="gray_r")

plt.tight_layout()
plt.show()
```

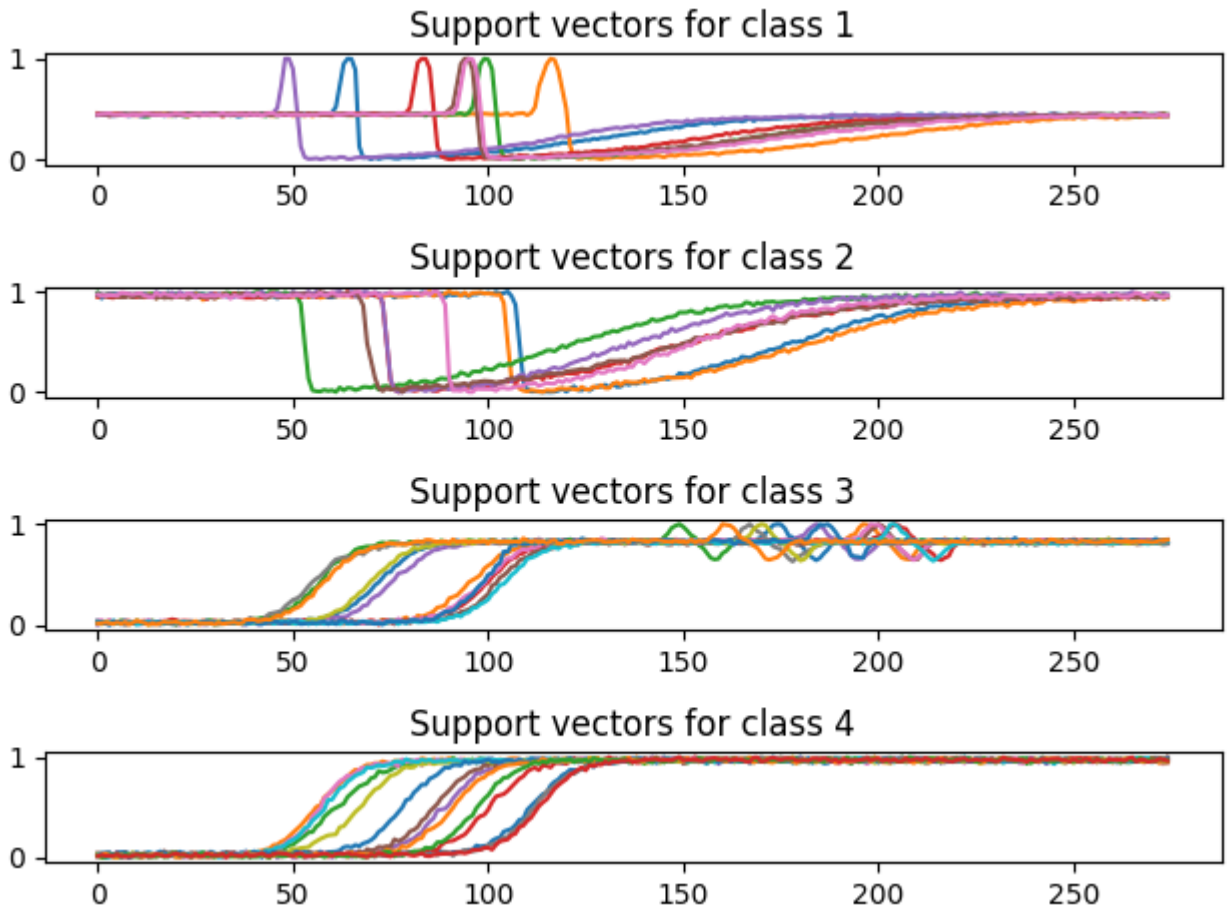
Total running time of the script: (0 minutes 1.569 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.2 SVM and GAK

This example illustrates the use of the global alignment kernel for support vector classification.

This metric is defined in the *tslearn.metrics* module and explained in details in "Fast global alignment kernels", by M. Cuturi (ICML 2011).



Out:

```
Correct classification rate: 1.0
```

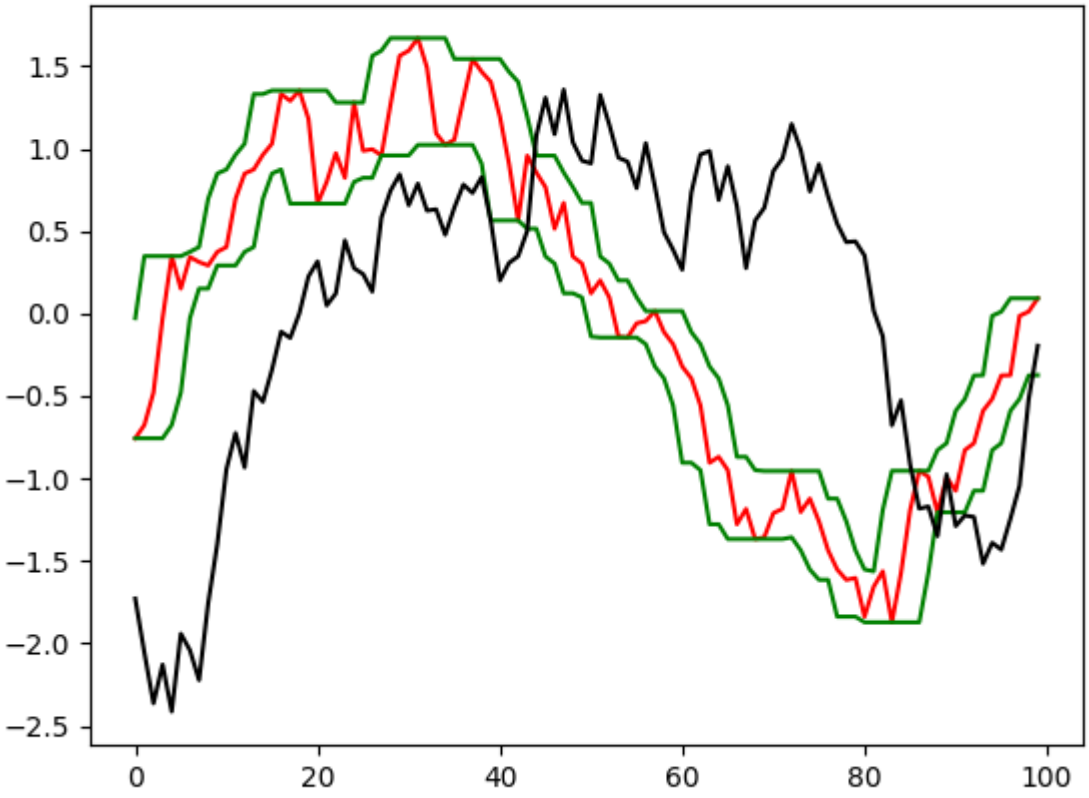
```
# Author: Romain Tavenard
# License: BSD 3 clause

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))

n_classes = len(set(y_train))

plt.figure()
support_vectors = clf.support_vectors_time_series_(X_train)
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 9.655 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.3 LB_Keogh

This example illustrates the principle of time series envelope as used in LB_Keogh and estimates similarity between time series using LB_Keogh.



Out:

```
LB_Keogh similarity:  10.321066157946282
```

```
# 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 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)
```

(continues on next page)

(continued from previous page)

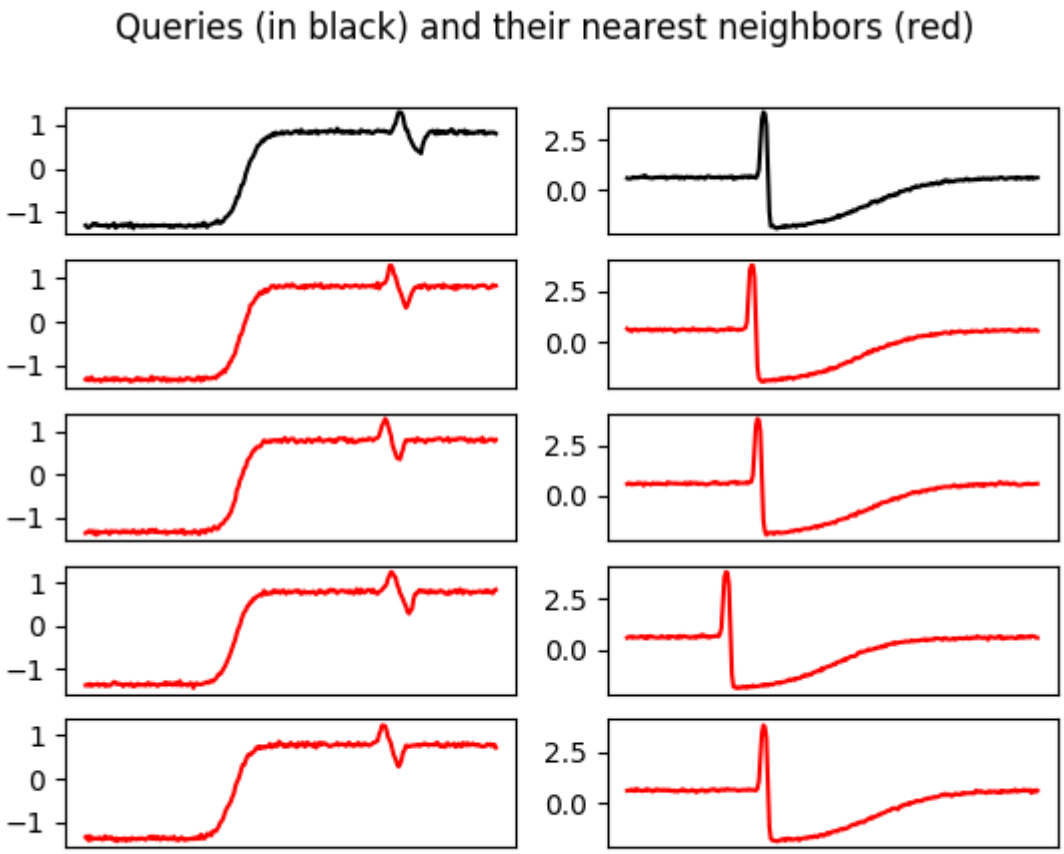
```
plt.figure()

envelope_down, envelope_up = metrics.lb_envelope(dataset_scaled[0], radius=3)
plt.plot(numpy.arange(sz), dataset_scaled[0, :, 0], "r-")
plt.plot(numpy.arange(sz), envelope_down[:, 0], "g-")
plt.plot(numpy.arange(sz), envelope_up[:, 0], "g-")
plt.plot(numpy.arange(sz), dataset_scaled[1, :, 0], "k-")

plt.show()

lb_k_sim = metrics.lb_keogh(dataset_scaled[1],
                            envelope_candidate=(envelope_down, envelope_up))

print("LB_Keogh similarity: ", lb_k_sim)
```

Total running time of the script: (0 minutes 1.063 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.4 k-NN search

This example performs a k-Nearest-Neighbor search in a database of time series using DTW as a base metric.




```
# 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
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-")
```

(continues on next page)

(continued from previous page)

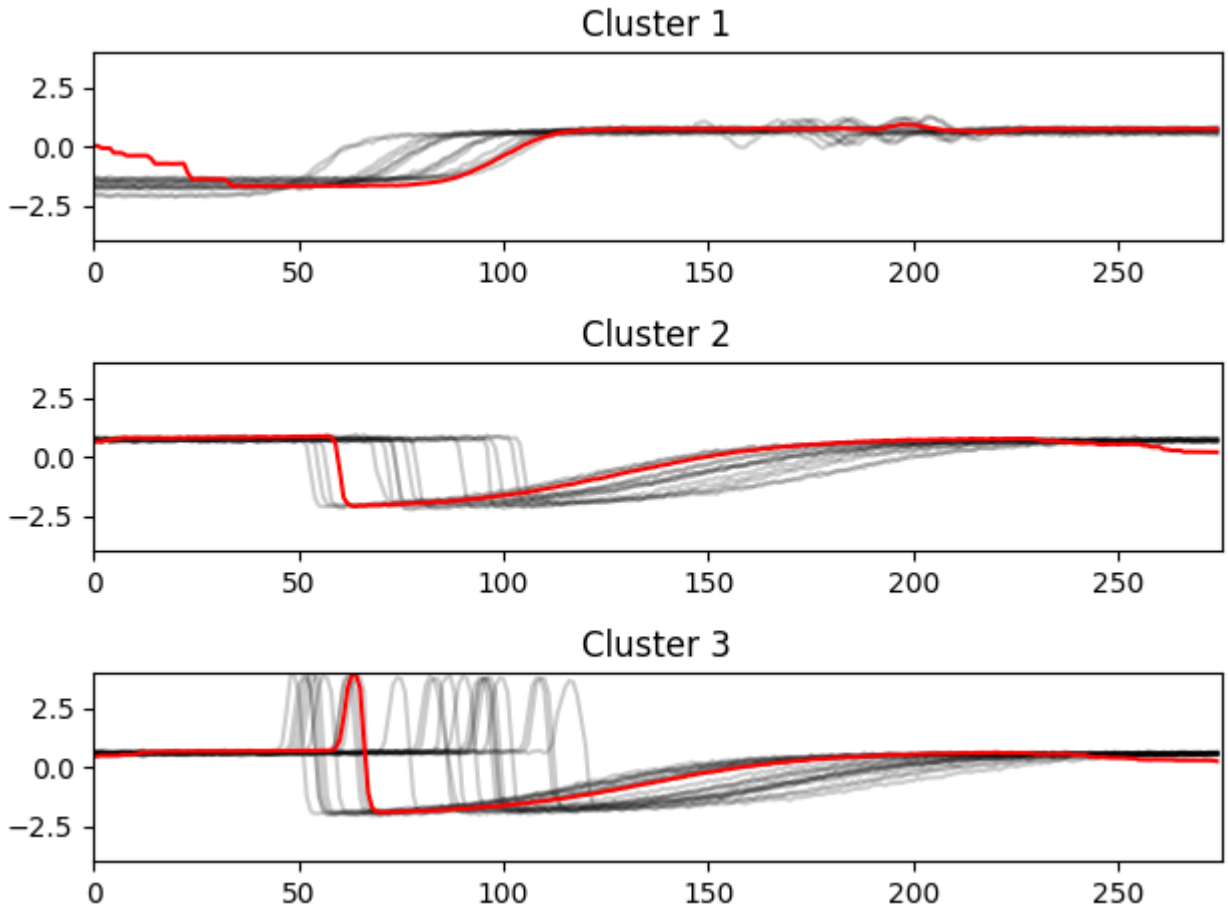
```
    plt.xticks([])
    for rank_nn in range(n_neighbors):
        plt.subplot(n_neighbors + 1, n_queries,
                    idx_ts + (n_queries * (rank_nn + 1)) + 1)
        plt.plot(X_train[ind[idx_ts, rank_nn]].ravel(), "r-")
        plt.xticks([])


plt.suptitle("Queries (in black) and their nearest neighbors (red)")
plt.show()
```

Total running time of the script: (0 minutes 1.158 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.5 KShape

This example uses the KShape clustering method that is based on cross-correlation to cluster time series.



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 -->
```

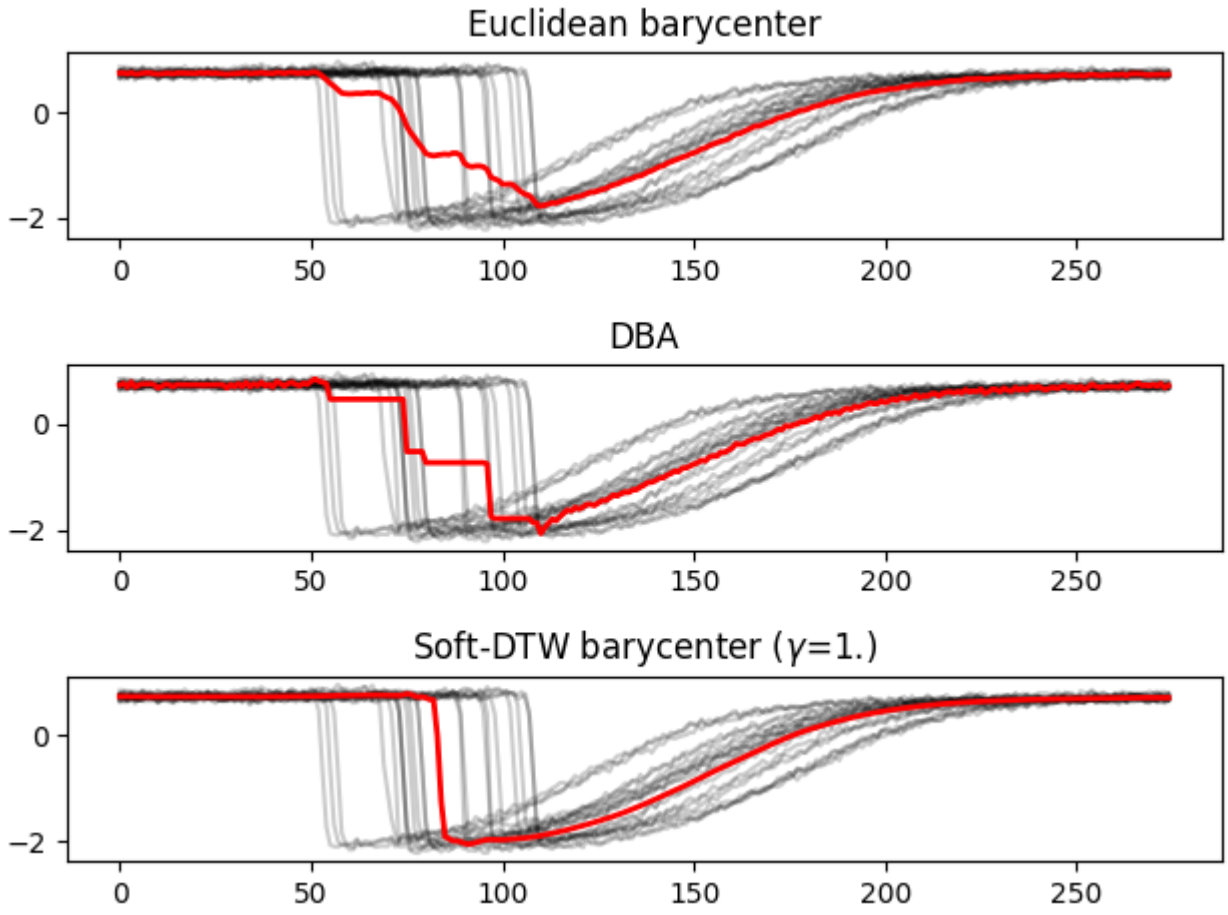
```
# Author: Romain Tavenard
# License: BSD 3 clause

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
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]

# Euclidean k-means
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)
    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 3.976 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.6 Barycenters

This example shows three methods to compute barycenters of time series.




```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.barycenters import euclidean_barycenter, \
    dtw_barycenter_averaging, softdtw_barycenter
from tslearn.datasets import CachedDatasets

numpy.random.seed(0)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
X = X_train[y_train == 2]

plt.figure()
plt.subplot(3, 1, 1)
for ts in X:
    plt.plot(ts.ravel(), "k-", alpha=.2)
plt.plot(euclidean_barycenter(X).ravel(), "r-", linewidth=2)
plt.title("Euclidean barycenter")
```

(continues on next page)

(continued from previous page)

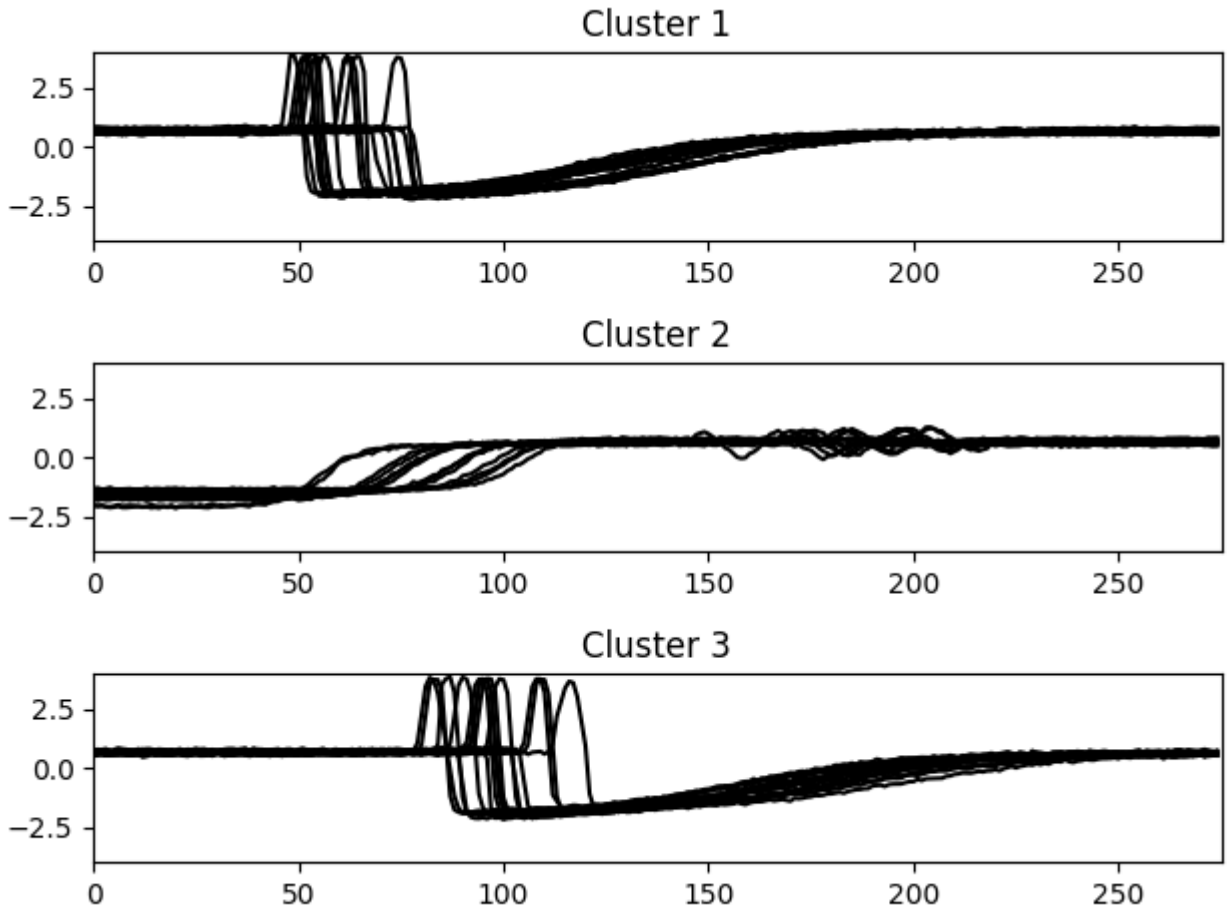
```
plt.subplot(3, 1, 2)
dba_bar = dtw_barycenter_averaging(X, max_iter=100, verbose=False)
for ts in X:
    plt.plot(ts.ravel(), "k-", alpha=.2)
plt.plot(dba_bar.ravel(), "r-", linewidth=2)
plt.title("DBA")

plt.subplot(3, 1, 3)
sdtw_bar = softdtw_barycenter(X, gamma=1., max_iter=100)
for ts in X:
    plt.plot(ts.ravel(), "k-", alpha=.2)
plt.plot(sdtw_bar.ravel(), "r-", linewidth=2)
plt.title("Soft-DTW barycenter ($\gamma$=1.)")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 8.987 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.7 Kernel k-means

This example uses Global Alignment kernel at the core of a kernel k-means algorithm to perform time series clustering.



Out:

```
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 -->
```

(continues on next page)

(continued from previous page)

```
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 --
↪>
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 GlobalAlignmentKernelKMeans
from tslearn.metrics import sigma_gak
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 = GlobalAlignmentKernelKMeans(n_clusters=3,
                                     sigma=sigma_gak(X_train),
                                     n_init=20,
                                     verbose=True,
                                     random_state=seed)
y_pred = gak_km.fit_predict(X_train)
```

(continues on next page)

(continued from previous page)

```
plt.figure()
for yi in range(3):
    plt.subplot(3, 1, 1 + yi)
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.title("Cluster %d" % (yi + 1))

plt.tight_layout()
plt.show()
```

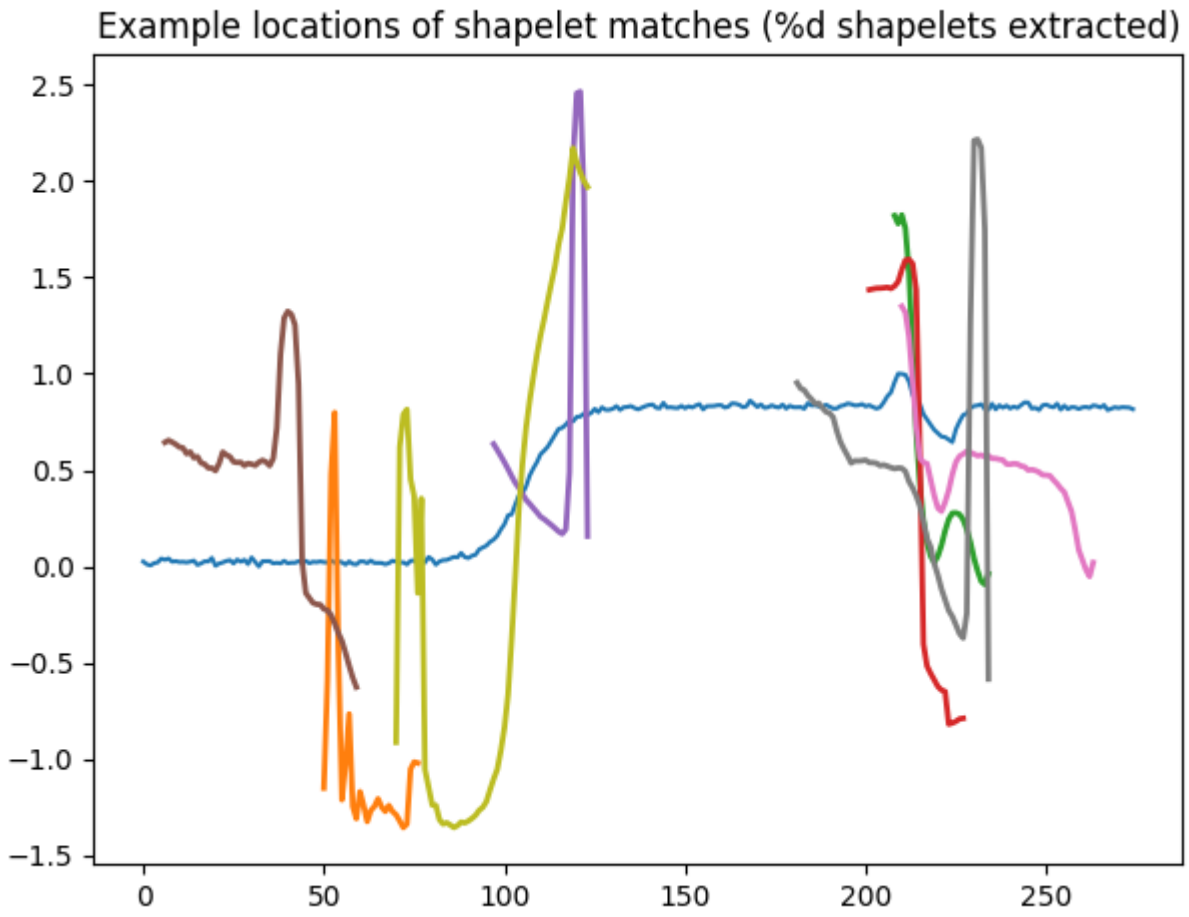
Total running time of the script: (0 minutes 6.479 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.8 Learning Shapelets

This example illustrates the use of the "Learning Shapelets" method for a time series classification task and `tslearn`'s shapelet localization method.

More information on the method can be found at: http://fs.ismll.de/publicspace/LearningShapelets/.




```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
from keras.optimizers import Adagrad
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.shapelets import ShapeletModel, \
    grabocka_params_to_shapelet_size_dict

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)

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,
```

(continues on next page)

(continued from previous page)

```
                                                        l=0.1,
                                                        r=2)

shp_clf = ShapeletModel(n_shapelets_per_size=shapelet_sizes,
                        optimizer=Adagrad(lr=.1),
                        weight_regularizer=.01,
                        max_iter=50,
                        verbose=0)
shp_clf.fit(X_train, y_train)
predicted_locations = shp_clf.locate(X_test)

test_ts_id = 0
plt.figure()
n_shapelets = sum(shapelet_sizes.values())
plt.title("Example locations of shapelet matches "
          "(%d shapelets extracted)".format(n_shapelets))
plt.plot(X_test[test_ts_id].ravel())
for idx_shp, shp in enumerate(shp_clf.shapelets_):
    t0 = predicted_locations[test_ts_id, idx_shp]
    plt.plot(numpy.arange(t0, t0 + len(shp)), shp, linewidth=2)

plt.tight_layout()
plt.show()
```

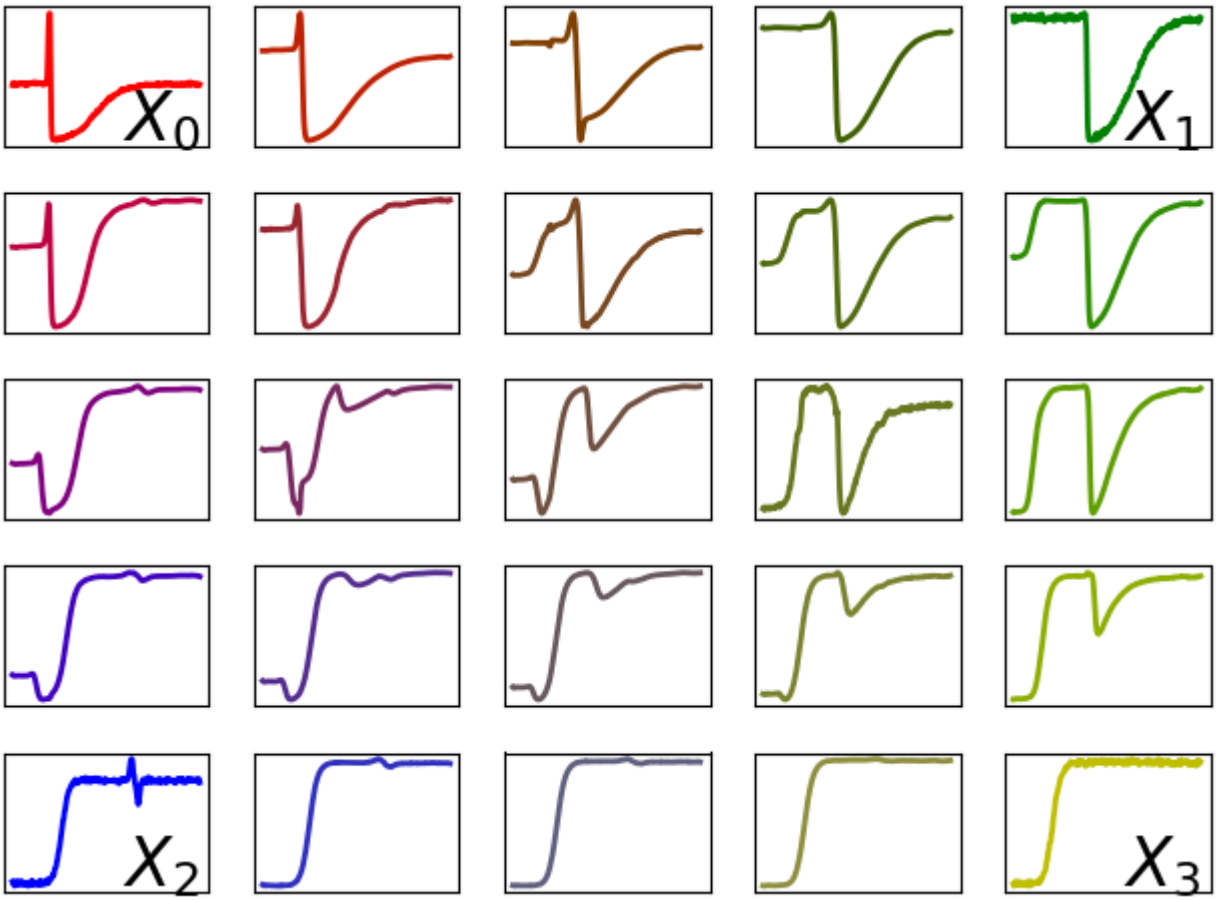
Total running time of the script: (0 minutes 5.597 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.9 Soft-DTW weighted barycenters

This example presents the weighted Soft-DTW time series barycenter method.

X_0, X_1, X_2 and X_3 are time series from 4 different classes in the Trace dataset. Other time series are weighted barycenters.




```
# 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()
```

(continues on next page)

(continued from previous page)

```
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
    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()
```

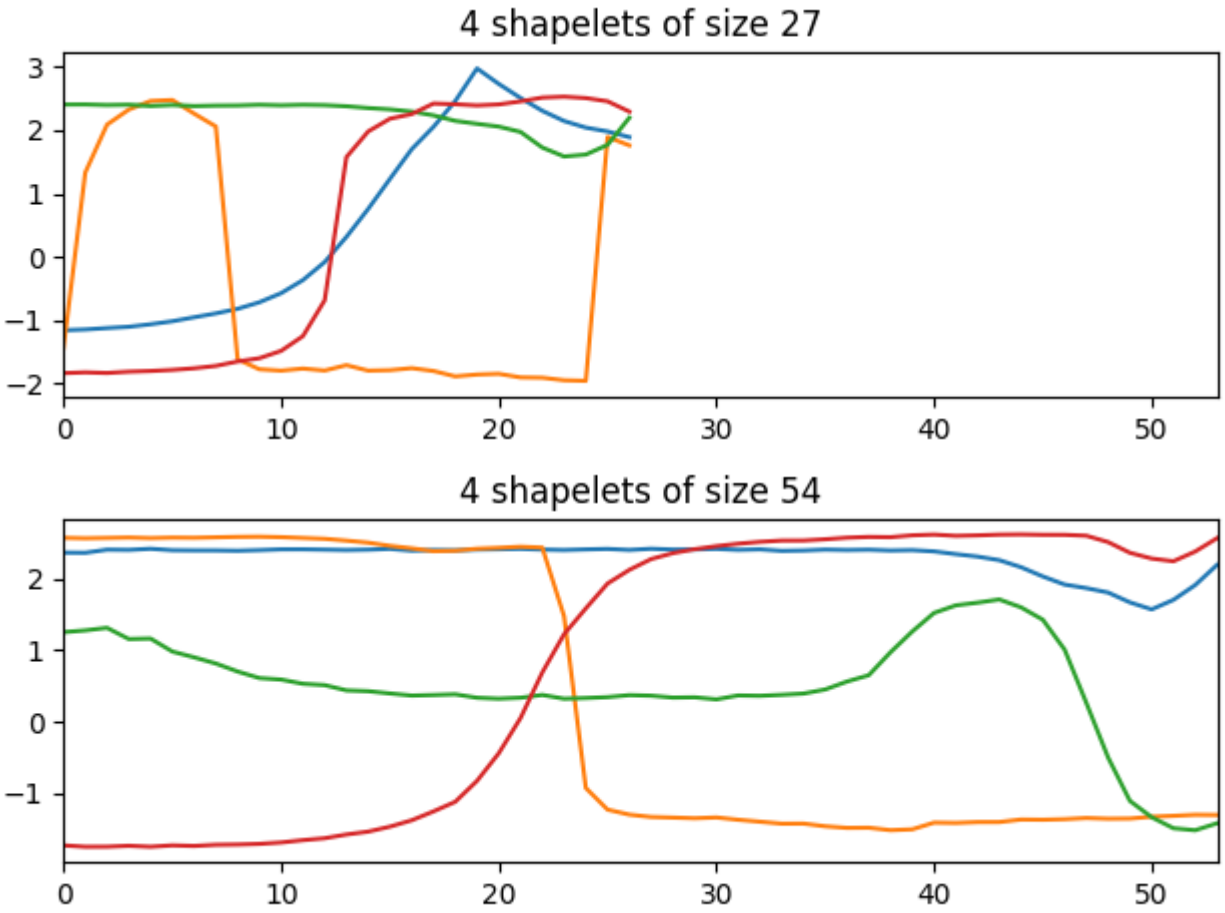
Total running time of the script: (0 minutes 5.913 seconds)

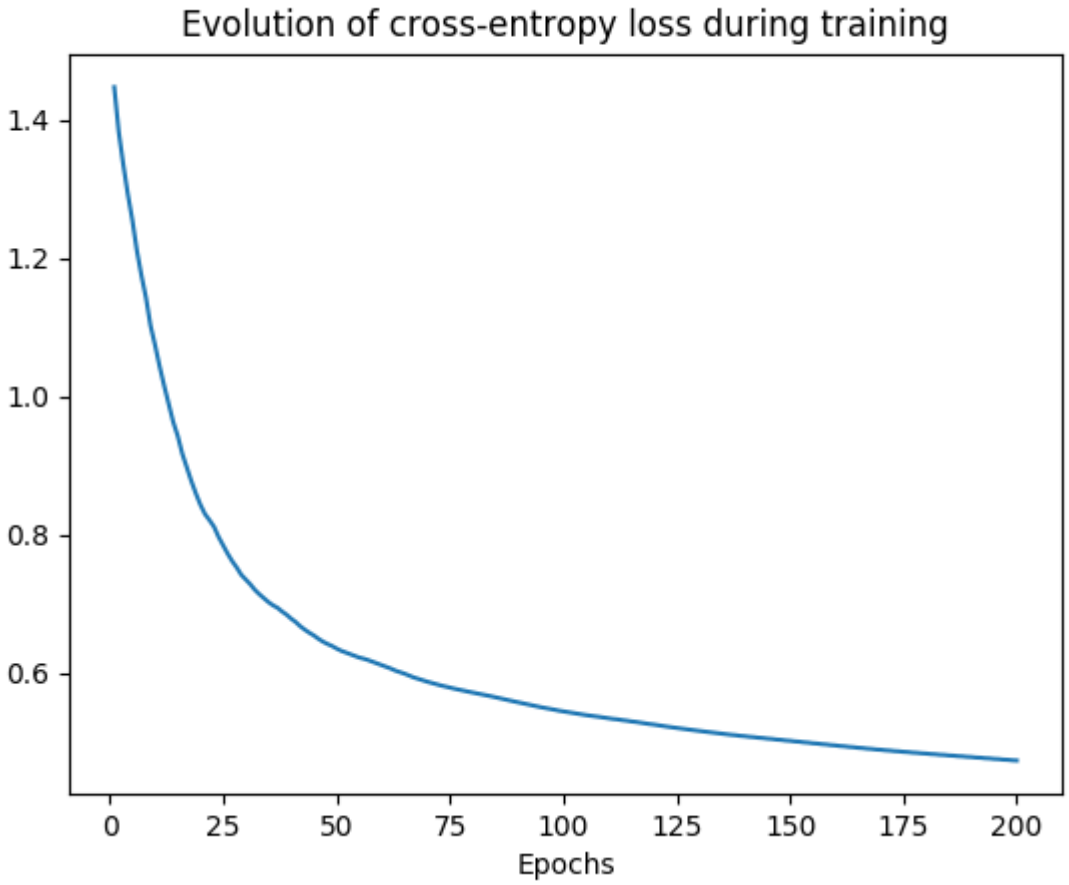
Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.10 Learning Shapelets

This example illustrates the use of the "Learning Shapelets" method for a time series classification task.

More information on the method can be found at: http://fs.ismll.de/publicspace/LearningShapelets/.


4 shapelets of size 27
3
2
1
0
−1
−2
0
10
20
30
40
50
4 shapelets of size 54
2
1
0
−1
0
10
20
30
40
50



Out:

```
Correct classification rate: 1.0
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
from sklearn.metrics import accuracy_score
from keras.optimizers import Adagrad
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.shapelets import ShapeletModel, \
    grabocka_params_to_shapelet_size_dict
from tslearn.utils import ts_size

numpy.random.seed(0)
```

(continues on next page)

(continued from previous page)

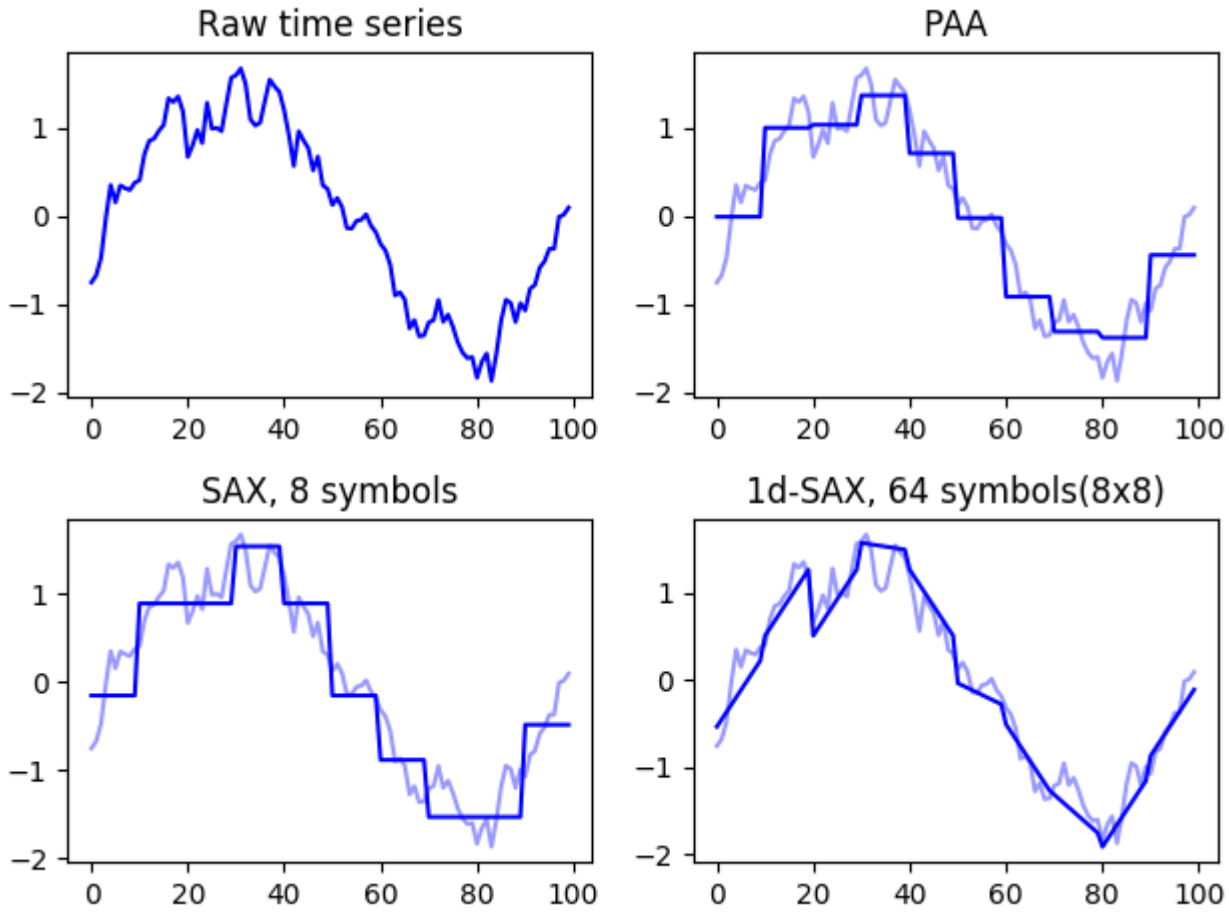
```
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
X_train = TimeSeriesScalerMinMax().fit_transform(X_train)
X_test = TimeSeriesScalerMinMax().fit_transform(X_test)

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=2)

# Define the model using parameters provided by the authors (except that we use
# fewer iterations here)
shp_clf = ShapeletModel(n_shapelets_per_size=shapelet_sizes,
                        optimizer=Adagrad(lr=.1),
                        weight_regularizer=.01,
                        max_iter=200,
                        verbose=0)
shp_clf.fit(X_train, y_train)
predicted_labels = shp_clf.predict(X_test)
print("Correct classification rate:", accuracy_score(y_test, predicted_labels))

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` attribute that is a keras
# model
plt.figure()
plt.plot(numpy.arange(1, 201), shp_clf.model_.history.history["loss"])
plt.title("Evolution of cross-entropy loss during training")
plt.xlabel("Epochs")
plt.show()
```

Total running time of the script: (0 minutes 9.964 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.11 PAA and SAX features

This example presents a comparison between PAA, SAX and 1d-SAX features.




```
# 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
n_paa_segments = 10
paa = PiecewiseAggregateApproximation(n_segments=n_paa_segments)
paa_dataset_inv = paa.inverse_transform(paa.fit_transform(dataset))
```

(continues on next page)

(continued from previous page)

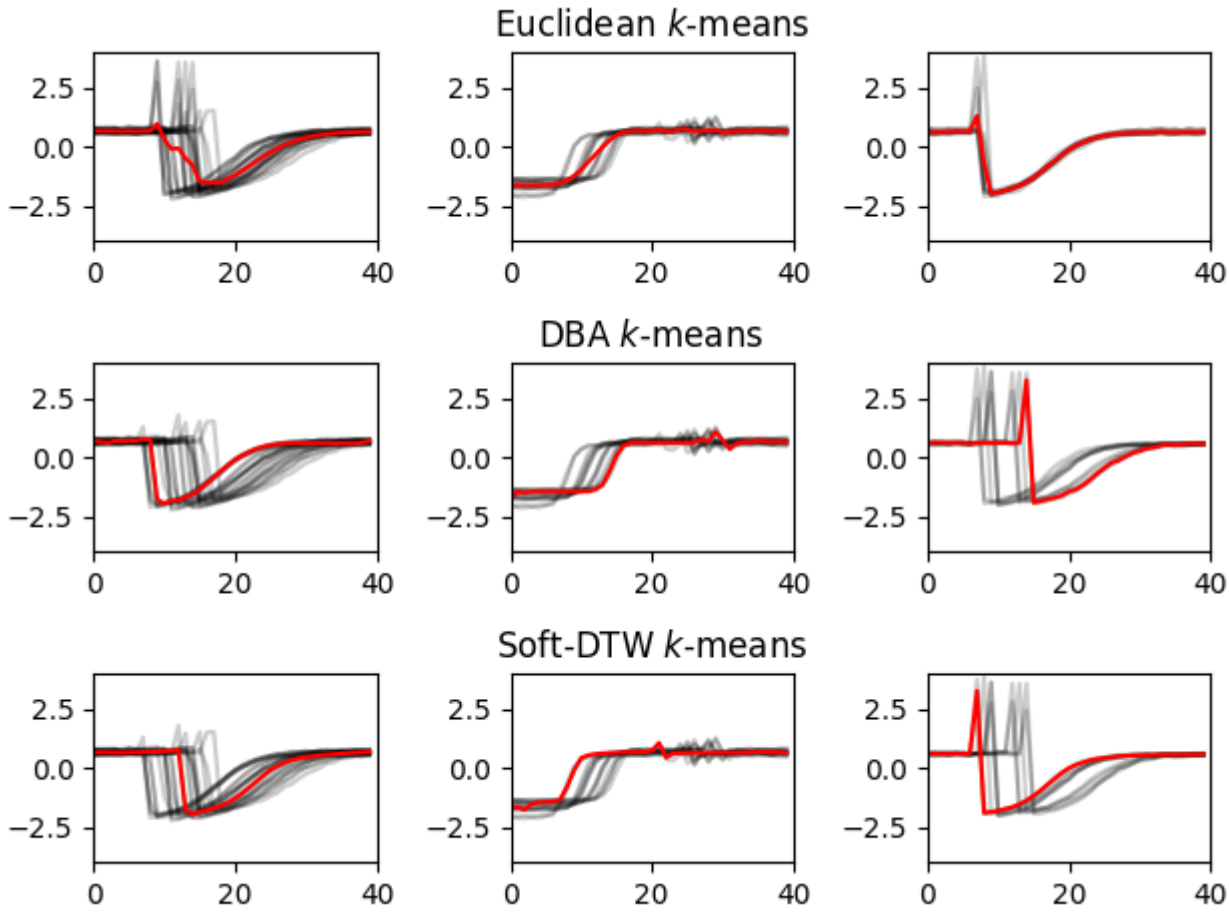
```
# SAX transform
n_sax_symbols = 8
sax = SymbolicAggregateApproximation(n_segments=n_paa_segments,
                                     alphabet_size_avg=n_sax_symbols)
sax_dataset_inv = sax.inverse_transform(sax.fit_transform(dataset))

# 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 0.449 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.12 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) and Soft-DTW k-means.



Out:

```
Euclidean k-means
16.434 --> 9.437 --> 9.437 -->
DBA k-means
Init 1
1.061 --> 0.473 --> 0.473 -->
Init 2
1.024 --> 0.528 --> 0.457 --> 0.452 --> 0.452 --> 0.452 -->
Soft-DTW k-means
/home/docs/checkouts/readthedocs.org/user_builds/tslearn/checkouts/latest/tslearn/
↪docs/examples/plot_kmeans.py:77: DeprecationWarning: 'gamma_sdtw' is deprecated in␣
↪version 0.2 and will be removed in 0.4. Use `gamma` instead of `gamma_sdtw` as a␣
↪`metric_params` key to remove this warning.
  random_state=seed)
2.475 --> 0.158 --> 0.157 --> 0.157 --> 0.157 --> 0.157 --> 0.157 --> 0.157 --> 0.157␣
↪--> 0.157 --> 0.157 --> 0.157 --> 0.157 --> 0.158 --> 0.157 --> 0.156 --> 0.156 -->␣
↪0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.
↪156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156␣
↪--> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 --> 0.156 -->␣
↪0.156 -->
```

```
# 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])
# 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)
    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)
    if yi == 1:
        plt.title("DBA $k$-means")
```

(continues on next page)

(continued from previous page)

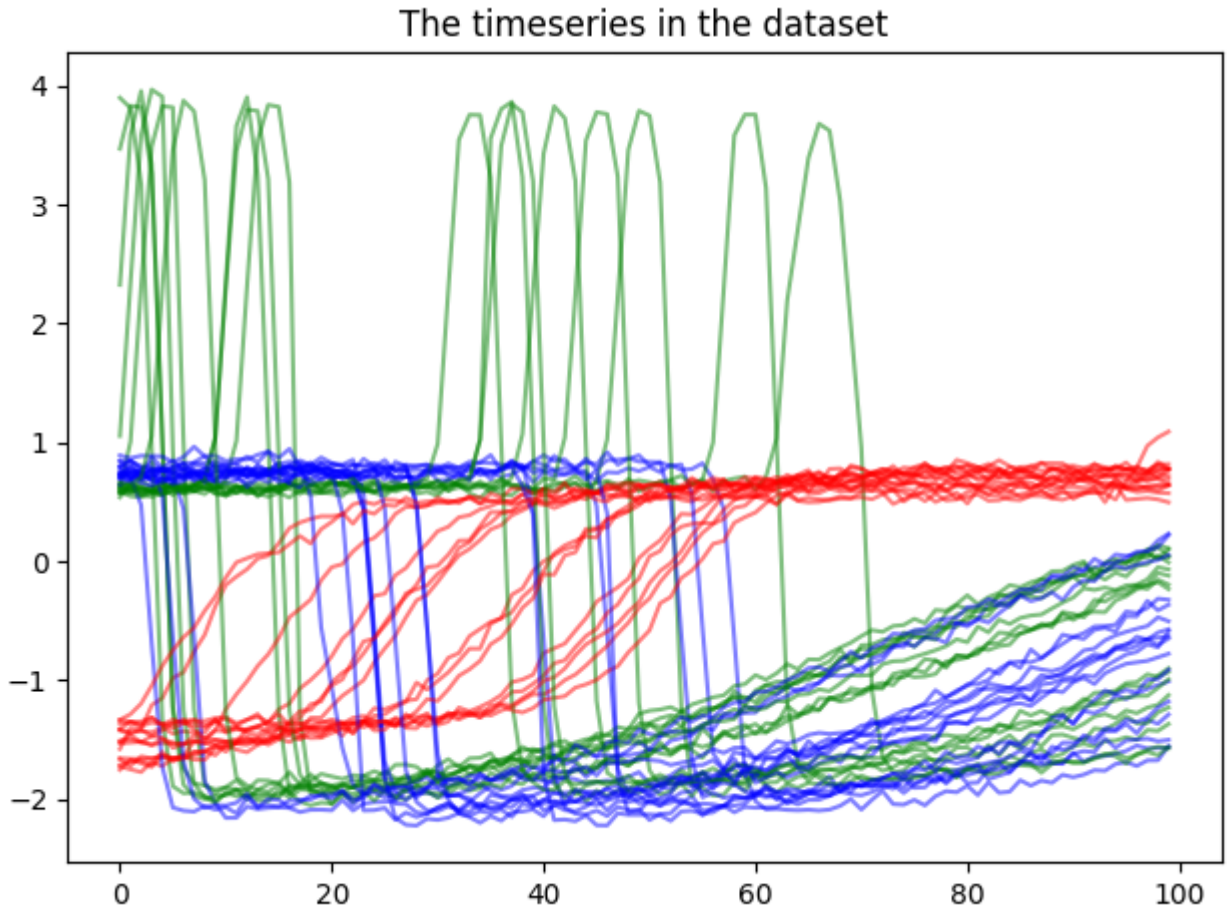
```
# Soft-DTW-k-means
print("Soft-DTW k-means")
sdtw_km = TimeSeriesKMeans(n_clusters=3,
                           metric="softdtw",
                           metric_params={"gamma_sdtw": .01},
                           verbose=True,
                           random_state=seed)
y_pred = sdtw_km.fit_predict(X_train)

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)
    plt.plot(sdtw_km.cluster_centers_[yi].ravel(), "r-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    if yi == 1:
        plt.title("Soft-DTW $k$-means")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 11.158 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.13 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*.



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|         1.0|     0.94118|      0.9375|
|           5|    distance|         1.0|         1.0|         1.0|
|          25|     uniform|     0.58824|     0.64706|       0.625|
|          25|    distance|     0.94118|     0.94118|      0.9375|
```

```
# Author: Gilles Vandewiele
# License: BSD 3 clause

from tslearn.neighbors import KNeighborsTimeSeriesClassifier
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.datasets import CachedDatasets
```

(continues on next page)

(continued from previous page)

```
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),
    iid=True
)

X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Keep only timeseries of class 0, 1 or 2
X_train = X_train[y_train < 4]
y_train = y_train[y_train < 4]

# 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 - 1], alpha=0.5)
plt.title('The timeseries in the dataset')
plt.tight_layout()
plt.show()

# Fit our pipeline
print('Performing hyper-parameter tuning of KNN classifier...')
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)
```

(continues on next page)

(continued from previous page)

```
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())
```

Total running time of the script: (0 minutes 3.934 seconds)

Note: Click *here* to download the full example code or run this example in your browser via Binder

2.4.14 Nearest neighbors

This example illustrates the use of nearest neighbor methods for database search and classification tasks.

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]
 [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: 1.0
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

from __future__ import print_function
import numpy
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline

from tslearn.generators import random_walk_blobs
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.neighbors import KNeighborsTimeSeriesClassifier, \
    KNeighborsTimeSeries
from tslearn.piecewise import SymbolicAggregateApproximation

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]
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")
```

(continues on next page)

(continued from previous page)

```
print("Correct classification rate:", accuracy_score(y_test, predicted_labels))

# Nearest neighbor classification  based on SAX representation
sax_trans = SymbolicAggregateApproximation(n_segments=10, alphabet_size_avg=5)
knn_clf = KNeighborsTimeSeriesClassifier(n_neighbors=3, metric="euclidean")
pipeline_model = Pipeline(steps=[('sax', sax_trans), ('knn', knn_clf)])
pipeline_model.fit(X_train, y_train)
predicted_labels = pipeline_model.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 0.542 seconds)

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

[R8d0e3400ad02-1] 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

[R25f7dec5cd6c-1] A. Bagnall, J. Lines, W. Vickers and E. Keogh, The UEA & UCR Time Series Classification Repository, www.timeseriesclassification.com

[R06dcc6755300-1] A. Bagnall, J. Lines, W. Vickers and E. Keogh, The UEA & UCR Time Series Classification Repository, www.timeseriesclassification.com

[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] 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.

[Rbedf9fcb0133-1] 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)

[R48c3dfce0d76-1] 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.

[R83f60d91f77e-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.

[R1290393aa4b0-1] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

[R9a1317388c84-1] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

Python Module Index

t

Index

G

I

K

L

O

P

R

S

T

U