tslearn Documentation

Release 0.9.0.dev0

Romain Tavenard

Jun 19, 2026

CONTENTS

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

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

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

CHAPTER

ONE

QUICK-START GUIDE

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

1.1 Installation

1.1.1 Using conda

The easiest way to install `tslearn` is probably via `conda` (preferably in a dedicated environment):

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

1.1.2 Using PyPI

Using `pip` should also work fine (preferably in a dedicated virtual environment):

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

1.1.3 Using latest github-hosted version

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

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

1.1.4 A note on requirements

`tslearn` builds on (and hence depends on) `scikit-learn`, `numpy` and `scipy` libraries. It also depends on the `numba` and `joblib` libraries.

Dependency	Version specifiers
scikit-learn	>=1.4
numpy	>=1.24.3
scipy	>=1.10.1
numba	>=0.61
joblib	>=1.2

Those should automatically be pulled on a standard `tslearn` installation.

If you plan to use the `tslearn.shapelets` module from `tslearn`, `keras` (v3+) and a dedicated backend should also be installed. See *the shapelets section* for more information.

`pytorch` can also be used as a computational backend for some metrics. See *the backend section* for more information.

`h5py` is required for reading or writing models using the hdf5 file format.

The `cesium` and `pandas` libraries may also be required if you plan on *integrating with some other python packages*.

You can use the *[all_features]* extra to enjoy all the features provided in the `tslearn` package:

```
python -m pip install tslearn[all_features]
```

1.2 Getting started

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

1.2.1 Time series format

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

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

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

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

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

1.2.2 Importing standard time series datasets

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

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

(continues on next page)

(continued from previous page)

```
>>> print(y_train.shape)
(1000,)
```

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

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

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

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

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

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

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

1.2.3 Playing with your data

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

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

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

1.3 Methods for variable-length time series

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

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

1.3.1 Classification

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

Examples

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

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

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

1.3.2 Regression

- *tslearn.svm.TimeSeriesSVR*

Examples

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

1.3.3 Nearest-neighbor search

- *tslearn.neighbors.KNeighborsTimeSeries*

Examples

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

1.3.4 Clustering

- *tslearn.clustering.KernelKMeans*
- *tslearn.clustering.TimeSeriesKMeans*
- *tslearn.clustering.TimeSeriesDBSCAN*
- *tslearn.clustering.silhouette_score*

Examples

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

```
from tslearn.clustering import TimeSeriesDBSCAN
dbscan = TimeSeriesDBSCAN(min_ts=2, eps=1, metric="dtw")
labels_dbscan = dbscan.fit_predict(X)
```

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

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

1.3.5 Barycenter computation

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

Examples

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

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

1.3.6 Preprocessing

- *tslearn.preprocessing.TimeSeriesScalerMinMax*
- *tslearn.preprocessing.TimeSeriesScalerMeanVariance*
- *tslearn.preprocessing.TimeSeriesImputer*

Examples

```
from tslearn.preprocessing import (
    TimeSeriesScalerMinMax,
    TimeSeriesScalerMeanVariance,
    TimeSeriesImputer
)
scaled = TimeSeriesScalerMinMax().fit_transform(X)
```

(continues on next page)

(continued from previous page)

```
scaled = TimeSeriesScalerMeanVariance().fit_transform(X)
imputed = TimeSeriesImputer().fit_transform(X)
```

1.3.7 Model selection

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

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

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

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

1.3.8 Resampling

- *tslearn.preprocessing.TimeSeriesResampler*

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

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

```
from tslearn.preprocessing import TimeSeriesResampler

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

1.4 Backend selection and use

`tslearn` proposes different backends (*NumPy* and *PyTorch*) to compute time series metrics such as *DTW* and *Soft-DTW*. The *PyTorch* backend can be used to compute gradients of metric functions thanks to automatic differentiation. The *PyTorch* backend requires the pytorch package to be installed..

1.4.1 Backend selection

A backend can be instantiated using the function `instantiate_backend`. To specify which backend should be instantiated (*NumPy* or *PyTorch*), this function accepts four different kind of input parameters:

- a string equal to `"numpy"` or `"pytorch"`.
- a *NumPy* array or a *Torch* tensor.
- a Backend instance. The input backend is then returned.
- `None` or anything else than mentioned previously. The backend *NumPy* is then instantiated.

Examples

If the input is the string "numpy", the NumPyBackend is instantiated.

```
>>> from tslearn.backend import instantiate_backend
>>> be = instantiate_backend("numpy")
>>> print(be.backend_string)
"numpy"
```

If the input is the string "pytorch", the PyTorchBackend is instantiated.

```
>>> be = instantiate_backend("pytorch")
>>> print(be.backend_string)
"pytorch"
```

If the input is a *NumPy* array, the NumPyBackend is instantiated.

```
>>> import numpy as np
>>> be = instantiate_backend(np.array([0]))
>>> print(be.backend_string)
"numpy"
```

If the input is a *Torch* tensor, the PyTorchBackend is instantiated.

```
>>> import torch
>>> be = instantiate_backend(torch.tensor([0]))
>>> print(be.backend_string)
"pytorch"
```

If the input is a Backend instance, the input backend is returned.

```
>>> print(be.backend_string)
"pytorch"
>>> be = instantiate_backend(be)
>>> print(be.backend_string)
"pytorch"
```

If the input is None, the NumPyBackend is instantiated.

```
>>> be = instantiate_backend(None)
>>> print(be.backend_string)
"numpy"
```

If the input is anything else, the NumPyBackend is instantiated.

```
>>> be = instantiate_backend("Hello, World!")
>>> print(be.backend_string)
"numpy"
```

The function instantiate_backend accepts any number of input parameters, including zero. To select which backend should be instantiated (*NumPy* or *PyTorch*), a for loop is performed on the inputs until a backend is selected.

```
>>> be = instantiate_backend(1, None, "Hello, World!", torch.tensor([0]), "numpy")
>>> print(be.backend_string)
"pytorch"
```

If none of the inputs are related to *NumPy* or *PyTorch*, the `NumPyBackend` is instantiated.

```
>>> be = instantiate_backend(1, None, "Hello, World!")
>>> print(be.backend_string)
"numpy"
```

1.4.2 Use the backends

The names of the attributes and methods of the backends are inspired by the *NumPy* backend.

Examples

Create backend objects.

```
>>> be = instantiate_backend("pytorch")
>>> mat = be.array([[0 , 1], [2, 3]], dtype=float)
>>> print(mat)
tensor([[0., 1.],
        [2., 3.]], dtype=torch.float64)
```

Use backend functions.

```
>>> norm = be.linalg.norm(mat)
>>> print(norm)
tensor(3.7417, dtype=torch.float64)
```

1.4.3 Choose the backend used by metric functions

tslearn's metric functions have an optional input parameter "`be`" to specify the backend to use to compute the metric.

Examples

```
>>> import torch
>>> from tslearn.metrics import dtw
>>> s1 = torch.tensor([[1.0], [2.0], [3.0]], requires_grad=True)
>>> s2 = torch.tensor([[3.0], [4.0], [-3.0]])
>>> sim = dtw(s1, s2, be="pytorch")
>>> print(sim)
sim tensor(6.4807, grad_fn=<SqrtBackward0>)
```

By default, the optional input parameter `be` is equal to `None`. Note that the first line of the function `dtw` is:

```
be = instantiate_backend(be, s1, s2)
```

Therefore, even if `be=None`, the `PyTorchBackend` is instantiated and used to compute the DTW metric since `s1` and `s2` are *Torch* tensors.

```
>>> sim = dtw(s1, s2)
>>> print(sim)
sim tensor(6.4807, grad_fn=<SqrtBackward0>)
```

1.4.4 Automatic differentiation

The *PyTorch* backend can be used to compute the gradients of the metric functions thanks to automatic differentiation.

Examples

Compute the gradient of the Dynamic Time Warping similarity measure.

```
>>> s1 = torch.tensor([[1.0], [2.0], [3.0]], requires_grad=True)
>>> s2 = torch.tensor([[3.0], [4.0], [-3.0]])
>>> sim = dtw(s1, s2, be="pytorch")
>>> sim.backward()
>>> d_s1 = s1.grad
>>> print(d_s1)
tensor([[-0.3086],
        [-0.1543],
        [ 0.7715]])
```

Compute the gradient of the Soft-DTW similarity measure.

```
>>> from tslearn.metrics import soft_dtw
>>> ts1 = torch.tensor([[1.0], [2.0], [3.0]], requires_grad=True)
>>> ts2 = torch.tensor([[3.0], [4.0], [-3.0]])
>>> sim = soft_dtw(ts1, ts2, gamma=1.0, be="pytorch", compute_with_backend=True)
>>> print(sim)
tensor(41.1876, dtype=torch.float64, grad_fn=<SelectBackward0>)
>>> sim.backward()
>>> d_ts1 = ts1.grad
>>> print(d_ts1)
tensor([[-4.0001],
        [-2.2852],
        [10.1643]])
```

1.5 Integration with other Python packages

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

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

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

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

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

1.5.1 scikit-learn

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

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

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

1.5.2 pyts

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

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

       [[1],
        [4]]])

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

1.5.3 seglearn

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

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

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

(continues on next page)

(continued from previous page)

```
       [2.]]),
       array([[1.],
       [4.],
       [3.]])], dtype=object)
```

1.5.4 stumpy

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

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

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

1.5.5 sktime

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

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

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

1.5.6 pyflux

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

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

1.5.7 tsfresh

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

```
>>> import pandas as pd
>>> from tslearn.utils import from_tsfresh_dataset, to_tsfresh_dataset
>>> df = pd.DataFrame([[0, 0, 1.0],
...                    [0, 1, 2.0],
...                    [1, 0, 1.0],
...                    [1, 1, 4.0],
...                    [1, 2, 3.0]], columns=['id', 'time', 'dim_0'])
>>> from_tsfresh_dataset(df)
array([[[ 1.],
    [ 2.],
    [nan]],

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

1.5.8 cesium

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

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

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

1.6 Contributing

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

Contributions are managed through GitHub Issues and Pull Requests.

We are welcoming contributions in the following forms:

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

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

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

1.6.1 More details on Pull requests

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

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

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

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

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

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

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

5. Push the changes to your GitHub account with:

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

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

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

Pull Request Checklist

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

- Follow the PEP8 Guidelines.
- If your pull request addresses an issue, please use the pull request title to describe the issue and mention the issue number in the pull request description. This will make sure a link back to the original issue is created.
- All public methods should have informative docstrings with sample usage presented as doctests when appropriate.
- Please prefix the title of your pull request with `[MRG]` (Ready for Merge), if the contribution is complete and ready for a detailed review. An incomplete contribution – where you expect to do more work before receiving a full review – should be prefixed `[WIP]` (to indicate a work in progress) and changed to `[MRG]` when it matures. WIPs may be useful to: indicate you are working on something to avoid duplicated work, request broad review of functionality or API, or seek collaborators. WIPs often benefit from the inclusion of a task list in the PR description.
- When adding additional functionality, provide at least one example script in the `tslearn/docs/examples/` folder. Have a look at other examples for reference. Examples should demonstrate why the new functionality is useful in practice and, if possible, compare it to other methods available in tslearn.
- Documentation and high-coverage tests are necessary for enhancements to be accepted. Bug-fixes or new features should be provided with non-regression tests. These tests verify the correct behavior of the fix or feature. In this manner, further modifications on the code base are granted to be consistent with the desired behavior. For the Bug-fixes case, at the time of the PR, this tests should fail for the code base in master and pass for the PR code.
- At least one paragraph of narrative documentation with links to references in the literature (with PDF links when possible) and the example.

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

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

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

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

```
$ pip install -e .[tests,all_features]
$ pytest --cov
```

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

```
$ pip install -e .[docs,all_features]
$ cd docs
$ make html
```

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

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

CHAPTER

TWO

USER GUIDE

2.1 Dynamic Time Warping

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

```
from tslearn.metrics import dtw, dtw_path

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

2.1.1 Optimization problem

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

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

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

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

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

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

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

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

2.1.2 Algorithmic solution

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

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

    C[0, 0] = 0.

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

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

2.1.3 Using a different ground metric

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

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

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

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

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

2.1.4 Properties

Dynamic Time Warping holds the following properties:

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

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

2.1.5 Additional constraints

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

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

The corresponding code would be:

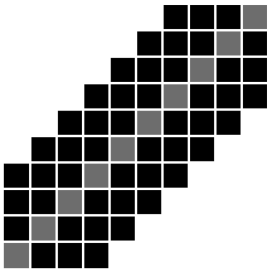


Fig. 1: $n = m = 10, r = 3$. Diagonal is marked in grey for better readability.

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

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

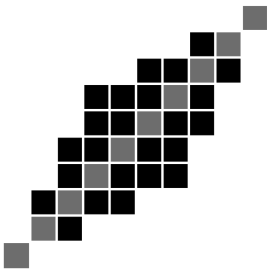


Fig. 2: $n = m = 10, s = 2$. Diagonal is marked in grey for better readability.

The corresponding code would be:

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

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

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

2.1.6 Barycenters

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

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

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

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

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

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

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

2.1.7 soft-DTW

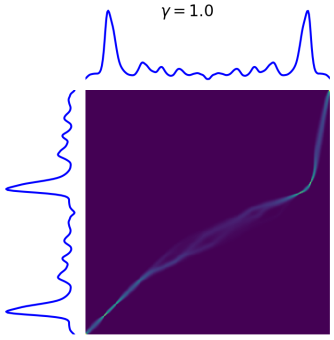
DTW is not differentiable with respect to its inputs because of the non-differentiability of the `min` operation. A differentiable extension has been presented in[4] in which the `min` operator is replaced by `soft-min`, using the log-sum-exp formulation:

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

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

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

When a strictly positive value is set for γ, the corresponding alignment matrix corresponds to a blurred version of the DTW one:



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

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

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

2.1.8 References

2.2 Longest Common Subsequence

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

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

[1] M. Vlachos, D. Gunopoulos, and G. Kollios. 2002. "Discovering Similar Multidimensional Trajectories", In Proceedings of the 18th International Conference on Data Engineering (ICDE '02). IEEE Computer Society, USA, 673.

```
from tslearn.metrics import lcss, lcss_path

lcss_score = lcss(x, y, eps=1)
# Or, if the path is also an important information:
path, lcss_score = lcss_path(x, y, eps=1)
```

2.2.1 Problem

The similarity S between x and y, given a positive real number ϵ, is formulated as follows:

$$S(x, y, \epsilon) = \frac{LCSS_\epsilon(x, y)}{\min(n, m)}$$

The constant ϵ is the matching threshold.

Here, a path can be seen as the parts of the time series where the Euclidean distance between them does not exceed a given threshold, i.e., they are close/similar.

To retrieve a meaningful similarity value from the length of the longest common subsequence, the percentage of that value regarding the length of the shortest time series is returned.

2.2.2 Algorithmic solution

There exists an $O(n^2)$ algorithm to compute the solution for this problem (pseudo-code is provided for time series indexed from 1 for simplicity):

```
def lcss(x, y):
   # Initialization
   for i = 0..n
       C[i, 0] = 0
   for j = 0..m
       C[0, j] = 0

   # Main loop
   for i = 1..n
        for j = 1..m
            if dist(x_i, x_j) <= epsilon:
                C[i, j] = C[i-1, j-1] + 1
            else:
                C[i, j] = max(C[i, j-1], C[i-1, j])

   return C[n, m] / min(n, m)
```

2.2.3 Using a different ground metric

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

```
from tslearn.metrics import lcss_path_from_metric
path, cost = lcss_path_from_metric(x, y, eps=1, metric=compatible_metric)
```

2.2.4 Properties

The Longest Common Subsequence holds the following properties:

- $\forall x, y, LCSS(x, y) \in [0, 1]$
- $\forall x, y, LCSS(x, y) = LCSS(y, x)$
- $\forall x, LCSS(x, x) = 1$

The values returned by LCSS range from 0 to 1, the value 1 being taken when the two time series completely match.

2.2.5 Additional constraints

One can set additional constraints to the set of acceptable paths. These constraints typically consists in forcing paths to lie close to the diagonal.

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

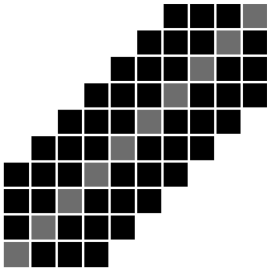


Fig. 3: $n = m = 10, r = 3$. Diagonal is marked in grey for better readability.

The corresponding code would be:

```
from tslearn.metrics import lcss
cost = lcss(x, y, eps=1, global_constraint="sakoe_chiba", sakoe_chiba_radius=3)
```

The Sakoe-Chiba radius corresponds to the parameter δ mentioned in[Page 20, 1], it controls how far in time we can go in order to match a given point from one time series to a point in another time series.

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

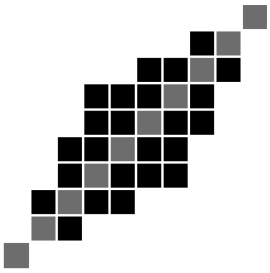


Fig. 4: $n = m = 10, s = 2$. Diagonal is marked in grey for better readability.

The corresponding code would be:

```
from tslearn.metrics import lcss
cost = lcss(x, y, eps=1, global_constraint="itakura", itakura_max_slope=2.)
```

2.2.6 Examples Involving LCSS variants

2.2.7 References

2.3 Kernel Methods

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

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

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

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

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

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

2.3.1 Global Alignment Kernel

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

The unnormalized GAK is defined, for a given bandwidth σ, as:

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

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

Note that the function `gak` is normalized in `tslearn`: it corresponds to the quotient

$$\text{gak}(\mathbf{x}, \mathbf{y}) = \frac{k(\mathbf{x}, \mathbf{y})}{\sqrt{k(\mathbf{x}, \mathbf{x}) k(\mathbf{y}, \mathbf{y})}}$$

This normalization ensures that $\text{gak}(\mathbf{x}, \mathbf{x}) = 1$ for all $\mathbf{x}$ and $\text{gak}(\mathbf{x}, \mathbf{y}) \in [0, 1]$ for all $\mathbf{x}, \mathbf{y}$.

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

```
from tslearn.metrics import gak, sigma_gak

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

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

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

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

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

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

2.3.2 Clustering and Classification

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

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

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

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

2.3.3 Examples Using Kernel Methods

2.3.4 References

2.4 Time Series Clustering

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

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

Fig. 5: k-means clustering with Euclidean distance. Each subfigure represents series from a given cluster and their centroid (in red).

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

2.4.1 k-means and Dynamic Time Warping

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

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

Fig. 6: k-means clustering with Dynamic Time Warping. Each subfigure represents series from a given cluster and their centroid (in red).

[3] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.
[2] I. S. Dhillon, Y. Guan & B. Kulis. "Kernel k-means, Spectral Clustering and Normalized Cuts," KDD 2004.

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

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

```
from tslearn.clustering import TimeSeriesKMeans

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

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

2.4.2 Kernel k-means and Time Series Kernels

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

Fig. 7: Kernel k-means clustering with Global Alignment Kernel. Each subfigure represents series from a given cluster.

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

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

2.4.3 Examples Using Clustering Estimators

2.4.4 References

2.5 Shapelets

Shapelets are defined in[1] as "subsequences that are in some sense maximally representative of a class". Informally, if we assume a binary classification setting, a shapelet is discriminant if it is **present** in most series of one class and absent from series of the other class. To assess the level of presence, one uses shapelet matches:

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

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

Fig. 8: The distance from a time series to a shapelet is done by sliding the shorter shapelet over the longer time series and calculating the point-wise distances. The minimal distance found is returned.

[1] M. Cuturi. "Fast Global Alignment Kernels," ICML 2011.
[2] I. S. Dhillon, Y. Guan & B. Kulis. "Kernel k-means, Spectral Clustering and Normalized Cuts," KDD 2004.
[3] H. Janati, M. Cuturi, A. Gramfort. "Spatio-Temporal Alignments: Optimal transport through space and time," AISTATS 2020
[4] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.
[1] L. Ye & E. Keogh. Time series shapelets: a new primitive for data mining. SIGKDD 2009.

In a classification setting, the goal is then to find the most discriminant shapelets given some labeled time series data. Shapelets can be mined from the training set[Page 25, 1] or learned using gradient-descent.

2.5.1 Learning Time-series Shapelets

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

Fig. 9: An example of how time series are transformed into linearly separable distances.

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

```
from tslearn.shapelets import LearningShapelets

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

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

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

2.5.2 Implementation note

The LearningShapelets implementation depends on `Keras` (v3+) which requires a dedicated backend. `keras` and `pytorch` backend are installed with `tslearn` through the [all_features] extra. You can install a Keras backend of your choice and use it by setting the KERAS_BACKEND environment variable. When KERAS_BACKEND environment variable is not set, the backend defaults to the first installed one among `pytorch`, `tensorflow` and `jax` in that order. Please note that setting the `Keras` backend through ~/.keras/keras.json is not supported.

2.5.3 References

2.6 Matrix Profile

The Matrix Profile, MP, is a new time series that can be calculated based on an input time series T and a subsequence length m. MP_i corresponds to the minimal distance from the query subsequence $T_{i \to i+m}$ to any subsequence in T[1].

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

[1] C. M. Yeh, Y. Zhu, L. Ulanova, N.Begum et al. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. ICDM 2016.

As the distance from the query subsequence to itself will be equal to zero, $T_{i-\frac{m}{4} \to i+\frac{m}{4}}$ is considered as an exclusion zone. In order to construct the Matrix Profile, a distance profile which is *similar to the distance calculation used to transform time series into their shapelet-transform space*, is calculated for each subsequence, as illustrated below:

Fig. 10: For each segment, the distances to all subsequences of the time series are calculated and the minimal distance that does not correspond to the original location of the segment (where the distance is zero) is returned.

2.6.1 Implementation

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

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

2.6.2 Possible Applications

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

2.6.3 References

2.7 Early Classification of Time Series

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

2.7.1 Early Classification Cost Function

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

The cost function is of the following form:

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

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

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

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

where:

[2] https://github.com/TDAmeritrade/stumpy

[3] https://www.cs.ucr.edu/~eamonn/MatrixProfile.html

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

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

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

Fig. 11: Early classification. At test time, prediction is made at a timestamp such that the expected earliness-accuracy is optimized, which can hence vary between time series.

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

```
from tslearn.early_classification import NonMyopicEarlyClassifier

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

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

2.7.2 References

CHAPTER

THREE

API REFERENCE

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

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

3.1 tslearn.barycenters

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

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

given data set x:

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

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

See the *barycenter examples* for an overview.

Functions

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

3.1.1 euclidean_barycenter

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

Standard Euclidean barycenter computed from a set of time series.

Parameters

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

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

Returns

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

Notes

This method requires a dataset of equal-sized time series

Examples

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

Examples using `tslearn.barycenters.euclidean_barycenter`

- *Barycenters*

3.1.2 dtw_barycenter_averaging

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

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

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

Parameters

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

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

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

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

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

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

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

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

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

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

References

[1], [2]

Examples

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

Examples using `tslearn.barycenters.dtw_barycenter_averaging`

- *Barycenters*

3.1.3 dtw_barycenter_averaging_subgradient

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

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

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

Parameters

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

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

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

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

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

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

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

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

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

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

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

Returns

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

References

[1], [2]

Examples

```
>>> time_series = [[1, 2, 3, 4], [1, 2, 4, 5]]
>>> dtw_barycenter_averaging_subgradient(
...     time_series,
...     max_iter=10,
...     random_state=0
```

(continues on next page)

(continued from previous page)

```
... )
array([[1. ],
       [2. ],
       [3.5...],
       [4.5...]])
```

Examples using `tslearn.barycenters.dtw_barycenter_averaging_subgradient`

- *Barycenters*

3.1.4 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.

Soft-DTW was originally presented in [1].

Parameters

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

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

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

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

tol: float
Tolerance of the method used.

max_iter: int
Maximum number of iterations.

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

Returns

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

References

[1]

Examples

```
>>> time_series = [[1, 2, 3, 4], [1, 2, 4, 5]]
>>> softdtw_barycenter(time_series, max_iter=5)
array([[1.25161574],
       [2.03821705],
```

(continues on next page)

(continued from previous page)

```
       [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`

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

3.2 tslearn.clustering

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

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

Classes

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

3.2.1 KernelKMeans

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

Kernel K-means.

Parameters

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

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

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

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

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

kernel_params
[dict or None (default: None)] Kernel parameters to be passed to the kernel function. None means no kernel parameter is set.

For Global Alignment Kernel, the only parameter of interest is *sigma*. If set to 'auto', it is computed based on a sampling of the training set (cf *tslearn.metrics.sigma_gak*). If no specific value is set for *sigma*, its defaults to 1. A *RuntimeError* is raised at fit time when computed or explicit value is close to 0 and therefore not compatible with 'gak' kernel.

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

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

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

Attributes

labels_
[numpy.ndarray] Labels of each point

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

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

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

Notes

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

References

[1], [2]

Examples

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

Methods

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

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

Compute kernel k-means clustering.

Parameters

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

y
Ignored

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

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

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

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

Parameters

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

y
Ignored

Returns

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

classmethod from_hdf5(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_json(path)`

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_pickle(path)`

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`predict(X)`

Predict the closest cluster each time series in X belongs to.

Parameters

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

Returns

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

set_fit_request(** (Keyword-only parameters separator (PEP 3102)), sample_weight: bool | None | str = '$UNCHANGED$'*) → *KernelKMeans*

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `fit`.

Returns

self
[object] The updated object.

set_params(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

to_hdf5(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

`to_json`(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

`to_pickle`(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

Examples using `tslearn.clustering.KernelKMeans`

- *Kernel k-means*

3.2.2 KShape

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

KShape clustering for time series.

KShape was originally presented in [1].

Parameters

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

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

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

n_init
[int (default: 1)] Number of time the k-Shape algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia. Ignored if initialization is not random.

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

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

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

Attributes

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

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

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

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

Notes

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

References

[1]

Examples

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

Methods

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

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

Compute k-Shape clustering.

Parameters

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

y
Ignored

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

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

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

Parameters

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

y
Ignored

Returns

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

`classmethod from_hdf5`(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_json`(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_pickle`(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Predict the closest cluster each time series in X belongs to.

Parameters

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

Returns

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

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`to_hdf5`(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

`to_json`(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

`to_pickle`(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

Examples using `tslearn.clustering.KShape`

- *KShape*
- *Model Persistence*

3.2.3 TimeSeriesKMeans

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

K-means clustering for time-series data.

Parameters

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

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

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

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

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

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

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

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

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

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

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

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

Attributes

labels_
[numpy.ndarray] Labels of each point.

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

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

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

Notes

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

Examples

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

(continues on next page)

(continued from previous page)

```
...                         metric="dtw", random_state=0).fit(X_bis)
>>> km.cluster_centers_.shape
(2, 6, 1)
```

Methods

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

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

Compute k-means clustering.

Parameters

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

y
Ignored

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

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

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

Parameters

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

y
Ignored

Returns

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

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

Fit to data, then transform it.

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

Parameters

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

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters. Pass only if the estimator accepts additional params in its *fit* method.

Returns

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

`classmethod from_hdf5`(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_json`(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_pickle`(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Predict the closest cluster each time series in X belongs to.

Parameters

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

Returns

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

`set_output`(**, transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`to_hdf5`(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

`to_json`(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

`to_pickle`(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

`transform`(*X*)

Transform X to a cluster-distance space.

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

Parameters

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

Returns

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

Examples using `tslearn.clustering.TimeSeriesKMeans`

- *k-means*

3.2.4 TimeSeriesDBSCAN

`class tslearn.clustering.TimeSeriesDBSCAN`(*eps=0.5*, *min_ts=5*, *metric='dtw'*, *metric_params=None*, *n_jobs=None*)

DBSCAN clustering for time series.

Parameters

eps
: [float (default: 0.5)] The maximum distance between two time series for one to be considered as in the neighborhood of the other.

min_ts
: [int (default: 5)] The number of time series (including itself) in a neighborhood for a time series to be considered as a core point.

metric: {'dtw', 'ctw', 'frechet', 'euclidean', 'precomputed'} (default: 'dtw')
: Metric to be used for similarity measure between time series.

metric_params
: [dict (default: None)] Additional keyword arguments to pass to the metric function. 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. Parameters that do not match the metric computation function signature are ignored.

n_jobs
: [int or None (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.

Attributes

core_ts_indices_
: [numpy.ndarray of shape (n_core_ts).] Indices of core time series.

components_: numpy.ndarray of shape (n_core_ts, sz, d)
: Copy of each core time series found by training.

labels_
: [numpy.ndarray of integers with shape (n_ts).] Labels of each time series. Noisy time series are given the label -1.

n_features_in_
: [int] Number of features seen during training.

Notes

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

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> from tslearn.preprocessing import TimeSeriesScalerMeanVariance
>>> X, y = random_walk_blobs(n_ts_per_blob=20, sz=32, d=2, n_blobs=4, random_
↪state=0)
>>> X = TimeSeriesScalerMeanVariance(mu=0., std=1.).fit_transform(X)
>>> db = TimeSeriesDBSCAN(eps=4, min_ts=3).fit(X)
>>> np.unique(db.labels_) # Clusters and noise
array([-1,  0,  1,  2,  3])
>>> list(db.labels_).count(-1) # Nb noisy elements
37
```

Methods

fit(X[, y])	Compute DBSCAN clustering.
fit_predict(X[, y])	Compute DBSCAN clustering.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.

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

Compute DBSCAN clustering.

Parameters

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

y
Ignored

Returns

TimeSeriesDBSCAN
The fitted estimator

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

Compute DBSCAN clustering.

Parameters

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

y
[Ignored] Not used, present here for API consistency by convention.

Returns

labels
[array of shape=(n_ts)] Index of the cluster each TS belongs to. Noisy TS are given the label -1.

classmethod from_hdf5(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_json`(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_pickle`(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

to_hdf5(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

to_json(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

to_pickle(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

Examples using `tslearn.clustering.TimeSeriesDBSCAN`

- *DBSCAN*

Functions

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

3.2.5 silhouette_score

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

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

Read more in the scikit-learn documentation.

Parameters

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

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

metric
[string, callable or None (default: None)] The metric to use when calculating distance between time series. Should be one of {'dtw', 'softdtw', 'euclidean'} or a callable distance function or None. If 'softdtw' is passed, a normalized version of Soft-DTW is used that is

defined as *sdtw_(x,y) := sdtw(x,y) - 1/2(sdtw(x,x)+sdtw(y,y))*. If X is the distance array itself, use `metric="precomputed"`. If None, dtw is used.

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

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

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

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

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

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

Returns

silhouette
[float] Mean Silhouette Coefficient for all samples.

References

[1], [2]

Examples

```
>>> from tslearn.generators import random_walks
>>> from tslearn.metrics import cdist_dtw
>>> from tslearn.metrics import dtw
>>> numpy.random.seed(0)
>>> X = random_walks(n_ts=20, sz=16, d=1)
>>> labels = numpy.random.randint(2, size=20)
>>> float(silhouette_score(X, labels, metric="dtw"))
0.13383800...
>>> float(silhouette_score(X, labels, metric="euclidean"))
0.09126917...
>>> float(silhouette_score(X, labels, metric="softdtw"))
0.17953934...
>>> float(silhouette_score(X, labels, metric="softdtw",
```

(continues on next page)

(continued from previous page)

```
...                          metric_params={"gamma": 2.}))
0.17591060...
>>> float(silhouette_score(cdist_dtw(X), labels,
...                          metric="precomputed"))
0.13383800...
>>> float(silhouette_score(X, labels, metric=dtw))
0.13383800...
```

3.3 tslearn.datasets

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

Note

MacOS users: If you encounter SSL certificate errors when downloading datasets, you may need to install certificates for your Python installation. Run the following command:

```
/Applications/Python<VERSION>/Install\ Certificates.command
```

Alternatively, install the `certifi` package:

```
pip install certifi
```

Classes

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

3.3.1 UCR_UEA_datasets

class tslearn.datasets.**UCR_UEA_datasets**(*use_cache=True*, *root_dir=None*)

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

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

This class will attempt to recover from some known misnamed files, like the *StarLightCurves* dataset being provided in *StarlightCurves.zip* and alike.

Parameters

use_cache
[bool (default: True)] Whether a cached version of the dataset should be used in *load_dataset()*, if one is found. Datasets are always cached upon loading, and this parameter only determines whether the cached version shall be refreshed upon loading.

root_dir
[str or None (default: None)] Directory to be used to cache downloaded datasets. If None, a default directory is used:

- If the environment variable *XDG_DATA_HOME* is set, the default directory is *$XDG_DATA_HOME/tslearn/UCR_UEA*.

- Otherwise, the default directory is *~/.tslearn/datasets/UCR_UEA*.

See also

CachedDatasets
Provides pre-selected datasets for offline use.

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 afterward.

References

[1]

Methods

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

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

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

Parameters

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

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

Returns

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

Examples

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

cache_all()

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

list_cached_datasets()

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

Examples

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

list_datasets()

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

Returns

list of str:
A list of names of all (univariate and multivariate) dataset names.

Examples

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

list_multivariate_datasets()

List multivariate datasets in the UCR/UEA archive.

Returns

list of str:
A list of the names of all multivariate dataset namas.

Examples

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

list_univariate_datasets()

List univariate datasets in the UCR/UEA archive.

Returns

list of str:
A list of the names of all univariate datasets.

Examples

```
>>> l = UCR_UEA_datasets().list_univariate_datasets()
>>> len(l)
128
```

load_dataset(*dataset_name*)

Load a dataset from the UCR/UEA archive from its name.

On failure, *None* is returned for each of the four values and a *RuntimeWarning* is printed.

Parameters

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

Returns

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

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

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

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

Examples

```
>>> data_loader = UCR_UEA_datasets()
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
...         "TwoPatterns")
>>> X_train.shape
(1000, 128, 1)
>>> y_train.shape
(1000,)
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
...         "Adiac")
>>> X_train.shape
(390, 176, 1)
>>> X_train, y_train, X_test, y_test = data_loader.load_dataset(
```

(continues on next page)

(continued from previous page)

```
...         "PenDigits")
>>> X_train.shape
(7494, 8, 2)
>>> assert (None, None, None, None) == data_loader.load_dataset(
...         "DatasetThatDoesNotExist")
```

Examples using `tslearn.datasets.UCR_UEA_datasets`

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

3.3.2 CachedDatasets

class tslearn.datasets.**CachedDatasets**

A convenience class to access cached time series datasets.

Note, that these *cached datasets* are statically included into *tslearn* and are distinct from the ones in *UCR_UEA_datasets*.

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

See also

UCR_UEA_datasets
: Provides more datasets and supports caching.

References

[1]

Methods

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

list_datasets()

List cached datasets.

Returns

list of str:
A list of names of all cached (univariate and multivariate) dataset names.

Examples

```
>>> from tslearn.datasets import CachedDatasets
>>> cached = CachedDatasets().list_datasets()
>>> "Trace" in cached
True
```

load_dataset(*dataset_name*)

Load a cached dataset from its name.

Parameters

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

Returns

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

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

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

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

Raises

IOError
If the dataset does not exist or cannot be read.

Examples

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

Examples using `tslearn.datasets.CachedDatasets`

- *k-NN search*
- *Hyper-parameter tuning of a pipeline with KNeighbors time series classifier*
- *DBSCAN*
- *Soft-DTW weighted barycenters*
- *Barycenters*
- *Kernel k-means*
- *k-means*
- *KShape*
- *Learning Shapelets: decision boundaries in 2D distance space*
- *Aligning discovered shapelets with timeseries*
- *Learning Shapelets*
- *SVM and GAK*
- *Soft-DTW loss for PyTorch neural network*

- *Distance and Matrix Profiles*
- *Model Persistence*

3.4 tslearn.early_classification

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

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

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

Classes

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

3.4.1 NonMyopicEarlyClassifier

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

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

Parameters

n_clusters
[int] Number of clusters to form.

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

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

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

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

random_state: int
Random state of the base estimator

Attributes

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

pyhatyck_
[array like of shape (maximum_time_stamp - min_t, n_cluster, __n_classes, __n_classes)] Contains the probabilities of being classified as class y_hat given class y and cluster ck for

a trained classifier. The penultimate dimension of the array is associated to the true class of the series and the last dimension to the predicted class.

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

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

References

[1]

Examples

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

Methods

early_classification_cost(X, y)	Compute early classification score.
early_predict(X)	Provides predicted classes as well as estimated delays before prediction timestamps for a dataset of incomplete time series.
early_predict_proba(X)	Provides probability estimates as well as estimated delays before prediction timestamps for a dataset of incomplete time series.
fit(X, y)	Fit early classifier.
get_cluster_probas(Xi)	Compute cluster probability $P(c_k \| Xi)$.
get_early_predict_generator([n_ts])	Allows streaming incoming timestamps of a time series and retrieving current predicted class as well as estimated delay before prediction timestamps.
get_early_predict_proba_generator([n_ts])	Allows streaming incoming timestamps of a time series and retrieving current probability estimates as well as estimated delay before prediction timestamps.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
predict(X)	Provide predicted class.
predict_class_and_earliness(X)	Provide predicted class as well as prediction timestamps.
predict_proba(X)	Probability estimates.
predict_proba_and_earliness(X)	Provide probability estimates as well as prediction timestamps.
score(X, y[, sample_weight])	Return accuracy on provided data and labels.
set_params(**params)	Set the parameters of this estimator.
set_score_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `score` method.

early_classification_cost(X, y)

Compute early classification score.

The score is computed as:

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

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

Parameters

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

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

Returns

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

Examples

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

early_predict(*X*)

Provides predicted classes as well as estimated delays before prediction timestamps for a dataset of incomplete time series.

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

Parameters

X
[array-like of shape (n_series, t, n_features)] A dataset of incomplete time series observed up to time t

Returns

array-like, shape (n_series,)
Predicted classes.

array-like, shape (n_series,)
Estimated delays before prediction timestamps.

early_predict_proba(*X*)

Provides probability estimates as well as estimated delays before prediction timestamps for a dataset of incomplete time series.

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

Parameters

X
[array-like of shape (n_series, t, n_features)] A dataset of incomplete time series observed up to time t

Returns

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

array-like of shape (n_series,)
Estimated delays before prediction timestamps.

`fit`(*X*, *y*)

Fit early classifier.

Parameters

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

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

Returns

self
[returns an instance of self.]

`get_cluster_probas`(*Xi*)

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

This quantity is computed using the following formula:

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

where

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

with

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

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

Parameters

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

Returns

probas
[numpy array, shape (n_clusters,)]

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 3, 2, 1],
```

(continues on next page)

(continued from previous page)

```
...                                     [1, 2, 3, 3, 2, 1],
...                                     [1, 2, 3, 3, 2, 1],
...                                     [3, 2, 1, 1, 2, 3],
...                                     [3, 2, 1, 1, 2, 3]])
>>> y = [0, 0, 0, 1, 1, 1, 0, 0]
>>> ts0 = to_time_series([1, 2])
>>> model = NonMyopicEarlyClassifier(n_clusters=3, lamb=0.,
...                                  random_state=0)
>>> probas = model.fit(dataset, y).get_cluster_probas(ts0)
>>> probas.shape
(3,)
>>> probas
array([0.33..., 0.33..., 0.33...])
>>> model = NonMyopicEarlyClassifier(n_clusters=3, lamb=10000.,
...                                  random_state=0)
>>> probas = model.fit(dataset, y).get_cluster_probas(ts0)
>>> probas.shape
(3,)
>>> probas
array([0.5, 0.5, 0. ])
>>> ts1 = to_time_series([3, 2])
>>> model.get_cluster_probas(ts1)
array([0., 0., 1.])
```

get_early_predict_generator(*n_ts=1*)

Allows streaming incoming timestamps of a time series and retrieving current predicted class as well as estimated delay before prediction timestamps.

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

Parameters

n_ts: int (default 1)
The number of time series that will be predicted by the generator

Returns

generator
Use the *send* method of the generator to stream timestamps as they become available and retrieve associated predictions and delays. This method takes an array-like, shape (n_ts, n_timestamps, n_features), representing freshly aquired data for all timeseries as input. This new data is concatenated with previously sent data, if any, to output the predicted classes and estimated delays before optimal prediction timestamps for all timeseries based all on available data (same output as *early_predict()*). Use n_timestamps = 1 for step by step feeding.

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
```

(continues on next page)

(continued from previous page)

```
...                                     [3, 2, 1, 1, 2, 3],
...                                     [3, 2, 1, 1, 2, 3]])
>>> y = [0, 0, 0, 1, 1, 1, 0, 0]
>>> model = NonMyopicEarlyClassifier(n_clusters=3, lamb=1000.,
...                                  cost_time_parameter=.1,
...                                  random_state=0).fit(dataset, y)
>>> incoming_timestamps = np.array([1, 2, 3, 3, 1, 2]).reshape(6, 1, 1, 1)
>>> gen = model.get_early_predict_generator()
>>> for x in incoming_timestamps:
...     print(gen.send(x))
(array([0]), array([3]))
(array([0]), array([2]))
(array([0]), array([1]))
(array([1]), array([0]))
(array([1]), array([0]))
(array([1]), array([0]))
```

get_early_predict_proba_generator(*n_ts=1*)

Allows streaming incoming timestamps of a time series and retrieving current probability estimates as well as estimated delay before prediction timestamps.

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

Parameters

n_ts: int (default 1)
The number of time series that will be predicted by the generator

Returns

generator
Use the *send* method of the generator to stream timestamps as they become available and retrieve associated prediction probalities and delays. This method takes an array-like, shape (n_ts, n_timestamps, n_features), representing freshly aquired data for all timeseries as input. This new data is concatenated with previously sent data, if any, to output the predicted probability estimates and estimated delays before optimal prediction timestamps for all timeseries based all on available data (same output as *early_predict_proba()*). Use n_timestamps = 1 for step by step feeding.

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 4, 5, 6],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [1, 2, 3, 3, 2, 1],
...                                   [3, 2, 1, 1, 2, 3],
...                                   [3, 2, 1, 1, 2, 3]])
>>> y = [0, 0, 0, 1, 1, 1, 0, 0]
>>> model = NonMyopicEarlyClassifier(n_clusters=3, lamb=1000.,
...                                  cost_time_parameter=.1,
...                                  random_state=0).fit(dataset, y)
>>> incoming_timestamps = np.array([1, 2, 3, 3, 1, 2]).reshape(6, 1, 1, 1)
```

(continues on next page)

(continued from previous page)

```
>>> gen = model.get_early_predict_proba_generator()
>>> for x in incoming_timestamps:
...     print(gen.send(x))
(array([[1., 0.]]), array([3]))
(array([[1., 0.]]), array([2]))
(array([[1., 0.]]), array([1]))
(array([[0., 1.]]), array([0]))
(array([[0., 1.]]), array([0]))
(array([[0., 1.]]), array([0]))
```

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

predict(*X*)

Provide predicted class.

Parameters

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

Returns

array, shape (n_series,)
Predicted classes.

predict_class_and_earliness(*X*)

Provide predicted class as well as prediction timestamps.

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

Parameters

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

Returns

array, shape (n_series,)
Predicted classes.

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

`predict_proba`(*X*)

Probability estimates.

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

Parameters

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

Returns

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

`predict_proba_and_earliness`(*X*)

Provide probability estimates as well as prediction timestamps.

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

Parameters

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

Returns

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

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

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

Return accuracy on provided data and labels.

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

Parameters

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

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

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

Returns

score
[float] Mean accuracy of `self.predict(X)` w.r.t. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *NonMyopicEarlyClassifier*

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

Returns

self
[object] The updated object.

Examples using `tslearn.early_classification.NonMyopicEarlyClassifier`

- *Early Classification*

3.5 tslearn.generators

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

Functions

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

3.5.1 random_walk_blobs

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

Blob-based random walk time series generator.

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

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

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

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

Parameters

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

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

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

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

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

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

Returns

numpy.ndarray
A dataset of random walk time series

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

Examples

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

Examples using `tslearn.generators.random_walk_blobs`

- *Nearest neighbors*

3.5.2 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 with a custom distance metric*
- *LB_Keogh*
- *Longest Common Subsequence*
- *Longest Common Subsequence with a custom distance metric*
- *sDTW multi path matching*
- *PAA and SAX features*

3.6 tslearn.matrix_profile

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

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

Classes

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

3.6.1 MatrixProfile

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

Matrix Profile transformation.

Matrix Profile was originally presented in [1].

Parameters

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

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

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

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

References

[1], [2]

Examples

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

Methods

fit(X[, y])	Fit a Matrix Profile representation.
fit_transform(X[, y])	Transform a dataset of time series into its Matrix Profile
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X[, y])	Transform a dataset of time series into its Matrix Profile

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

Fit a Matrix Profile representation.

Parameters

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

Returns

MatrixProfile
self

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

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

Parameters

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

Returns

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

`classmethod from_hdf5`(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_json`(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_pickle`(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`set_output`(*, transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`to_hdf5`(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

`to_json`(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

`to_pickle`(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

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

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

Parameters

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

Returns

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

Examples using `tslearn.matrix_profile.MatrixProfile`

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

3.7 tslearn.metrics

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

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

Functions

compute_mask(s1, s2[, global_constraint, ...])	Compute the mask (region constraint).
sakoe_chiba_mask(sz1, sz2[, radius, be])	Compute the Sakoe-Chiba mask.
itakura_mask(sz1, sz2[, max_slope, be])	Compute the Itakura mask.
cdist_dtw(dataset1[, dataset2, ...])	Compute cross-similarity matrix using Dynamic Time Warping (DTW) similarity measure.
cdist_gak(dataset1[, dataset2, sigma, ...])	Compute cross-similarity matrix using Global Alignment kernel (GAK).
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.
cdist_frechet(dataset1[, dataset2, ...])	Compute cross-similarity matrix using Frechet similarity measure [1].
ctw(s1, s2[, max_iter, n_components, ...])	Compute Canonical Time Warping (CTW) similarity measure between (possibly multidimensional) time series and return the similarity.
ctw_path(s1, s2[, max_iter, n_components, ...])	Compute Canonical Time Warping (CTW) similarity measure between (possibly multidimensional) time series and return the alignment path, the canonical correlation analysis (sklearn) object and the similarity.
dtw(s1, s2[, global_constraint, ...])	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series and return it.

continues on next page

Table 17 – continued from previous page

dtw_path(s1, s2[, global_constraint, ...])	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series and return both the path and the similarity.
dtw_path_from_metric(s1[, s2, metric, ...])	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series using a distance metric defined by the user and return both the path and the similarity.
dtw_accumulated_matrix(s1, s2, mask[, be])	Compute the DTW accumulated cost matrix score between two time series.
dtw_limited_warping_length(s1, s2, max_length)	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series under an upper bound constraint on the resulting path length and return the similarity cost.
dtw_path_limited_warping_length(s1, s2, ...)	Compute Dynamic Time Warping (DTW) similarity measure between (possibly multidimensional) time series under an upper bound constraint on the resulting path length and return the path as well as the similarity cost.
subsequence_path(acc_cost_mat, idx_path_end)	Compute the optimal path through an accumulated cost matrix given the endpoint of the sequence.
subsequence_cost_matrix(subseq, longseq[, be])	Compute the accumulated cost matrix score between a subsequence and a reference time series.
dtw_subsequence_path(subseq, longseq[, be])	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.
lcss(s1, s2[, eps, global_constraint, ...])	Compute the Longest Common Subsequence (LCSS) similarity measure between (possibly multidimensional) time series and return the similarity.
lcss_path(s1, s2[, eps, global_constraint, ...])	Compute the Longest Common Subsequence (LCSS) similarity measure between (possibly multidimensional) time series and return both the path and the similarity.
lcss_path_from_metric(s1[, s2, eps, metric, ...])	Compute the Longest Common Subsequence (LCSS) similarity measure between (possibly multidimensional) time series using a distance metric defined by the user and return both the path and the similarity.
gak(s1, s2[, sigma, be])	Compute Global Alignment Kernel (GAK) between (possibly multidimensional) time series and return it.
soft_dtw(ts1, ts2[, gamma, be, ...])	Compute Soft-DTW metric between two time series.
soft_dtw_alignment(ts1, ts2[, gamma, be, ...])	Compute Soft-DTW metric between two time series and return both the similarity measure and the alignment matrix.
lb_envelope(ts[, radius, be])	Compute time series envelope as required by LB_Keogh.
lb_keogh(ts_query[, ts_candidate, radius, ...])	Compute LB_Keogh.
sigma_gak(dataset[, n_samples, random_state, be])	Compute sigma value to be used for GAK.
gamma_soft_dtw(dataset[, n_samples, ...])	Compute gamma value to be used for GAK/Soft-DTW.
SoftDTWLossPyTorch([gamma, normalize, dist_func])	Soft-DTW loss function in PyTorch.
frechet(s1, s2[, global_constraint, ...])	Compute Frechet similarity [1] measure between (possibly multidimensional) time series and return it.

continues on next page

Table 17 – continued from previous page

frechet_path(s1, s2[, global_constraint, ...])	Compute Frechet similarity measure [1] between (possibly multidimensional) time series and an optimal alignment path.
frechet_path_from_metric(s1[, s2, metric, ...])	Compute Frechet similarity measure and an optimal alignment path [1] between (possibly multidimensional) time series using a distance metric defined by the user.
frechet_accumulated_matrix(s1, s2, mask[, be])	Compute the Frechet accumulated cost matrix score between two time series.

3.7.1 compute_mask

tslearn.metrics.**compute_mask**(*s1*, *s2*, *global_constraint=0*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *be=None*)

Compute the mask (region constraint).

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,) or int] A time series or integer. If shape is (sz1,), the time series is assumed to be univariate. If int, size sz1 used to dimension the mask.

s2
[array-like, shape=(sz2, d) or (sz2,) or int] Another time series or integer. If shape is (sz2,), the time series is assumed to be univariate. If int, size sz2 used to dimension the mask.

global_constraint
[{0, 1, 2} (default: 0)] Global constraint to restrict admissible paths for DTW: - "itakura" if 1 - "sakoe_chiba" if 2 - no constraint otherwise

sakoe_chiba_radius
[int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [1], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to 2 (sakoe-chiba), a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

itakura_max_slope
[float or None (default: None)] Maximum slope for the Itakura parallelogram constraint. If None and *global_constraint* is set to 1 (itakura), a maximum slope of 2. is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

mask
[array-like, shape=(sz1, sz2)] Constraint region.

References

[1]

Examples

```
>>> compute_mask(4, 4)
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
>>> compute_mask(4, 4, sakoe_chiba_radius=1)
array([[ True,  True, False, False],
       [ True,  True,  True, False],
       [False,  True,  True,  True],
       [False, False,  True,  True]])
>>> compute_mask(4, 4, itakura_max_slope=2)
array([[ True, False, False, False],
       [False,  True,  True, False],
       [False,  True,  True, False],
       [False, False, False,  True]])
```

3.7.2 sakoe_chiba_mask

tslearn.metrics.**sakoe_chiba_mask**(*sz1*, *sz2*, *radius=1*, *be=None*)

Compute the Sakoe-Chiba mask.

Parameters

sz1
[int] The size of the first time series

sz2
[int] The size of the second time series.

radius
[int] The radius of the band.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

mask
[array-like, shape=(sz1, sz2)] Sakoe-Chiba mask.

Examples

```
>>> sakoe_chiba_mask(4, 4, radius=2)
array([[ True,  True,  True, False],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [False,  True,  True,  True]])
```

3.7.3 itakura_mask

tslearn.metrics.**itakura_mask**(*sz1*, *sz2*, *max_slope=2.0*, *be=None*)

Compute the Itakura mask.

Parameters

sz1
[int] The size of the first time series

sz2
[int] The size of the second time series.

max_slope
[float (default = 2)] The maximum slope of the parallelogram.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

mask
[array-like, shape=(sz1, sz2)] Itakura mask.

Examples

```
>>> itakura_mask(6, 6, max_slope=3)
array([[ True, False, False, False, False, False],
       [False,  True,  True,  True, False, False],
       [False,  True,  True,  True,  True, False],
       [False,  True,  True,  True,  True, False],
       [False, False,  True,  True,  True, False],
       [False, False, False, False, False,  True]])
```

3.7.4 cdist_dtw

tslearn.metrics.**cdist_dtw**(*dataset1*, *dataset2=None*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *n_jobs=None*, *verbose=0*, *be=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 π is the alignment path:

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

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

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

Parameters

dataset1
[array-like, shape=(n_ts1, sz1, d) or (n_ts1, sz1) or (sz1,)] A dataset of time series. If shape is (n_ts1, sz1), the dataset is composed of univariate time series. If shape is (sz1,), the dataset is composed of a unique univariate time series.

dataset2
[None or array-like, shape=(n_ts2, sz2, d) or (n_ts2, sz2) or (sz2,) (default: None)] Another dataset of time series. If *None*, self-similarity of *dataset1* is returned. If shape is (n_ts2, sz2), the dataset is composed of univariate time series. If shape is (sz2,), the dataset is composed of a unique univariate time series.

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

sakoe_chiba_radius
[int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [1], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

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

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

cdist
[array-like, shape=(n_ts1, n_ts2)] Cross-similarity matrix.

See also

dtw
: Get DTW similarity score

References

[1]

Examples

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

3.7.5 cdist_gak

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

Compute cross-similarity matrix using Global Alignment kernel (GAK). Note that GAK is a kernel, the larger GAK values mean more similar time series.

GAK was originally presented in [1].

Parameters

dataset1
: [array-like, shape=(n_ts1, sz1, d) or (n_ts1, sz1) or (sz1,)] A dataset of time series. If shape is (n_ts1, sz1), the dataset is composed of univariate time series. If shape is (sz1,), the dataset is composed of a unique univariate time series.

dataset2
: [None or array-like, shape=(n_ts2, sz2, d) or (n_ts2, sz2) or (sz2,) (default: None)] Another dataset of time series. If *None*, self-similarity of *dataset1* is returned. If shape is (n_ts2, sz2), the dataset is composed of univariate time series. If shape is (sz2,), the dataset is composed of a unique univariate time series.

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

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

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

be
: [Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend*

or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

array-like, shape=(n_ts1, n_ts2)
Cross-similarity matrix.

See also

gak
Compute Global Alignment kernel

References

[1]

Examples

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

3.7.6 cdist_soft_dtw

`tslearn.metrics.`**`cdist_soft_dtw`**(*dataset1*, *dataset2=None*, *gamma=1.0*, *be=None*, *compute_with_backend=False*)

Compute cross-similarity matrix using Soft-DTW metric.

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

Soft-DTW is computed as:

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

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

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

Parameters

dataset1
[array-like, shape=(n_ts1, sz1, d) or (n_ts1, sz1) or (sz1,)] A dataset of time series. If shape is (n_ts1, sz1), the dataset is composed of univariate time series. If shape is (sz1,), the dataset is composed of a unique univariate time series.

dataset2
[None or array-like, shape=(n_ts2, sz2, d) or (n_ts2, sz2) or (sz2,) (default: None)] Another dataset of time series. If *None*, self-similarity of *dataset1* is returned. If shape is (n_ts2, sz2), the dataset is composed of univariate time series. If shape is (sz2,), the dataset is composed of a unique univariate time series.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

compute_with_backend
[bool, default=False] This parameter has no influence when the NumPy backend is used. When a backend different from NumPy is used (cf parameter *be*): If *True*, the computation is done with the corresponding backend. If *False*, a conversion to the NumPy backend can be used to accelerate the computation.

Returns

array-like, shape=(n_ts1, n_ts2)
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.        ]])
```

The PyTorch backend can be used to compute gradients:

```
>>> import torch
>>> dataset1 = torch.tensor([[[1.0], [2.0], [3.0]], [[1.0], [2.0], [3.0]]],␣
↪requires_grad=True)
>>> dataset2 = torch.tensor([[[3.0], [4.0], [-3.0]], [[3.0], [4.0], [-3.0]]])
>>> sim_mat = cdist_soft_dtw(dataset1, dataset2, gamma=1.0, be="pytorch", compute_
```

(continues on next page)

(continued from previous page)

```
↪with_backend=True)
>>> print(sim_mat)
tensor([[41.1876, 41.1876],
        [41.1876, 41.1876]], grad_fn=<CopySlices>)
>>> sim = sim_mat[0, 0]
>>> sim.backward()
>>> print(dataset1.grad)
tensor([[[-4.0001],
         [-2.2852],
         [10.1643]],

        [[ 0.0000],
         [ 0.0000],
         [ 0.0000]]])
```

3.7.7 cdist_soft_dtw_normalized

tslearn.metrics.**cdist_soft_dtw_normalized**(*dataset1*, *dataset2=None*, *gamma=1.0*, *be=None*, *compute_with_backend=False*)

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

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

Soft-DTW is computed as:

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

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

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

This normalized version is defined as:

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

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

Parameters

dataset1
: [array-like, shape=(n_ts1, sz1, d) or (n_ts1, sz1) or (sz1,)] A dataset of time series. If shape is (n_ts1, sz1), the dataset is composed of univariate time series. If shape is (sz1,), the dataset is composed of a unique univariate time series.

dataset2
: [None or array-like, shape=(n_ts2, sz2, d) or (n_ts2, sz2) or (sz2,) (default: None)] Another dataset of time series. If *None*, self-similarity of *dataset1* is returned. If shape is (n_ts2, sz2), the dataset is composed of univariate time series. If shape is (sz2,), the dataset is composed of a unique univariate time series.

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

be
: [Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

compute_with_backend
: [bool, default=False] This parameter has no influence when the NumPy backend is used. When a backend different from NumPy is used (cf parameter *be*): If *True*, the computation is done with the corresponding backend. If *False*, a conversion to the NumPy backend can be used to accelerate the computation.

Returns

array-like, shape=(n_ts1, n_ts2)
: 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 = np.random.randn(10, 15, 1)
>>> bool(np.all(cdist_soft_dtw_normalized(time_series) >= 0.))
True
>>> time_series2 = np.random.randn(4, 15, 1)
>>> bool(np.all(cdist_soft_dtw_normalized(time_series, time_series2) >= 0.))
True
```

The PyTorch backend can be used to compute gradients:

```
>>> import torch
>>> dataset1 = torch.tensor([[[1.0], [2.0], [3.0]], [[1.0], [2.0], [3.0]]],␣
↪requires_grad=True)
>>> dataset2 = torch.tensor([[[3.0], [4.0], [-3.0]], [[3.0], [4.0], [-3.0]]])
>>> sim_mat = cdist_soft_dtw_normalized(dataset1, dataset2, gamma=1.0, be="pytorch",
↪ compute_with_backend=True)
>>> print(sim_mat)
tensor([[42.0586, 42.0586],
        [42.0586, 42.0586]], grad_fn=<AddBackward0>)
>>> sim = sim_mat[0, 0]
>>> sim.backward()
>>> print(dataset1.grad)
tensor([[[-3.5249],
         [-2.2852],
```

(continues on next page)

(continued from previous page)

```
         [ 9.6891]],

        [[ 0.0000],
         [ 0.0000],
         [ 0.0000]]])
```

3.7.8 cdist_frechet

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

Compute cross-similarity matrix using Frechet similarity measure [1].

Frechet is computed as the maximum distance between aligned time series, i.e., if π is an optimal alignment path:

$$Frechet(X, Y) = \max_{(i,j)\in\pi} \|X_i - Y_j\|$$

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

It is not required that time series share the same size, but they must be the same dimension.

Parameters

dataset1
[array-like, shape=(n_ts1, sz1, d) or (n_ts1, sz1) or (sz1,)] A dataset of time series. If shape is (n_ts1, sz1), the dataset is composed of univariate time series. If shape is (sz1,), the dataset is composed of a unique univariate time series.

dataset2
[None or array-like, shape=(n_ts2, sz2, d) or (n_ts2, sz2) or (sz2,) (default: None)] Another dataset of time series. If *None*, self-similarity of *dataset1* is returned. If shape is (n_ts2, sz2), the dataset is composed of univariate time series. If shape is (sz2,), the dataset is composed of a unique univariate time series.

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

sakoe_chiba_radius
[int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [2], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

n_jobs
[int or None, optional (default=None)] The number of jobs to run in parallel. `None` means

1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

cdist
[array-like, shape=(n_ts1, n_ts2)] Cross-similarity matrix.

See also

frechet
Get only the similarity score

frechet_path
Get both the matching path and the similarity score

frechet_path_from_metric
Compute Frechet similarity score and path using a user-defined distance metric

References

[1], [2]

Examples

```
>>> cdist_frechet([[1, 2, 2, 3], [1., 2., 3., 4.]])
array([[0., 1.],
       [1., 0.]])
>>> cdist_frechet([[1, 2, 2, 3], [1., 2., 3., 4.]], [[1, 2, 3], [2, 3, 4, 5]])
array([[0., 2.],
       [1., 1.]])
```

3.7.9 ctw

`tslearn.metrics.ctw`(*s1*, *s2*, *max_iter=100*, *n_components=None*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *verbose=False*, *be=None*)

Compute Canonical Time Warping (CTW) similarity measure between (possibly multidimensional) time series and return the similarity.

Canonical Time Warping is a method to align time series under rigid registration of the feature space. It should not be confused with Dynamic Time Warping (DTW), though CTW uses DTW.

It is not required that both time series share the same size, nor the same dimension (CTW will find a subspace that best aligns feature spaces). CTW was originally presented in [1].

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

max_iter
[int (default: 100)] Number of iterations for the CTW algorithm. Each iteration

n_components
[int (default: None)] Number of components to be used for Canonical Correlation Analysis. If None, the lower minimum number of features between seq1 and seq2 is used.

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

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

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

verbose
[bool (default: True)] If True, scores are printed at each iteration of the algorithm.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

float
Similarity score

See also

ctw
Get only the similarity score for CTW

References

[1]

Examples

```
>>> float(ctw([1, 2, 3], [1., 2., 2., 3.]))
0.0
>>> float(ctw([1, 2, 3], [[1., 1.], [2., 2.], [2., 2.], [3., 3.]]))
0.0
```

3.7.10 ctw_path

tslearn.metrics.**ctw_path**(*s1*, *s2*, *max_iter=100*, *n_components=None*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *verbose=False*, *be=None*)

Compute Canonical Time Warping (CTW) similarity measure between (possibly multidimensional) time series and return the alignment path, the canonical correlation analysis (sklearn) object and the similarity.

Canonical Time Warping is a method to align time series under rigid registration of the feature space. It should not be confused with Dynamic Time Warping (DTW), though CTW uses DTW.

It is not required that both time series share the same size, nor the same dimension (CTW will find a subspace that best aligns feature spaces). CTW was originally presented in [1].

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

max_iter
[int (default: 100)] Number of iterations for the CTW algorithm. Each iteration

n_components
[int (default: None)] Number of components to be used for Canonical Correlation Analysis. If None, the lower minimum number of features between s1 and s2 is used.

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

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

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

verbose
: [bool (default: True)] If True, scores are printed at each iteration of the algorithm.

be
: [Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

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

sklearn.decomposition.CCA
: The Canonical Correlation Analysis object used to align time series at convergence.

float
: Similarity score

See also

ctw
: Get only the similarity score for CTW

References

[1]

Examples

```
>>> path, cca, dist = ctw_path([1, 2, 3], [1., 2., 2., 3.])
>>> path
[(0, 0), (1, 1), (1, 2), (2, 3)]
>>> type(cca)
<class 'sklearn.cross_decomposition...CCA'>
>>> float(dist)
0.0
>>> path, cca, dist = ctw_path([1, 2, 3],
...                            [[1., 1.], [2., 2.], [2., 2.], [3., 3.]])
>>> float(dist)
0.0
```

Examples using `tslearn.metrics.ctw_path`

- *Canonical Time Warping*

3.7.11 dtw

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

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

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

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

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

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

Parameters

s1
: [array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
: [array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

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

sakoe_chiba_radius
: [int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [1], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

be
: [Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

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
```

The PyTorch backend can be used to compute gradients:

```
>>> import torch
>>> s1 = torch.tensor([[1.0], [2.0], [3.0]], requires_grad=True)
>>> s2 = torch.tensor([[3.0], [4.0], [-3.0]])
>>> sim = dtw(s1, s2, be="pytorch")
>>> print(sim)
tensor(6.4807, dtype=torch.float64, grad_fn=<SqrtBackward0>)
>>> sim.backward()
>>> print(s1.grad)
tensor([[-0.3086],
        [-0.1543],
        [ 0.7715]])
```

```
>>> s1_2d = torch.tensor([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]], requires_grad=True)
>>> s2_2d = torch.tensor([[3.0, 3.0], [4.0, 4.0], [-3.0, -3.0]])
>>> sim = dtw(s1_2d, s2_2d, be="pytorch")
>>> print(sim)
tensor(9.1652, dtype=torch.float64, grad_fn=<SqrtBackward0>)
>>> sim.backward()
>>> print(s1_2d.grad)
tensor([[-0.2182, -0.2182],
        [-0.1091, -0.1091],
        [ 0.5455,  0.5455]])
```

3.7.12 dtw_path

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

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

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

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

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

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

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

sakoe_chiba_radius
[int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [1], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

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

float
Similarity score

See also

dtw
Get only the similarity score for DTW

cdist_dtw
Cross similarity matrix between time series datasets

dtw_path_from_metric
Compute a DTW using a user-defined distance metric

References

[1]

Examples

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

Examples using `tslearn.metrics.dtw_path`

- *Canonical Time Warping*
- *Dynamic Time Warping*
- *Frechet*
- *Longest Common Subsequence*

3.7.13 dtw_path_from_metric

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

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

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

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

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

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,) if metric!="precomputed", (sz1, sz2) otherwise] A time series or an array of pairwise distances between samples. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,), optional (default: None)] A second time series, only allowed if metric != "precomputed". If shape is (sz2,), the time series is assumed to be univariate.

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

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

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

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

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

sakoe_chiba_radius
[int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [1], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

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

Returns

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

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

See also

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

Notes

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

References

[1]

Examples

Lets create 2 numpy arrays to wrap:

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

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

```
>>> x, y = dtw_path_from_metric(s1, s2,
...                             metric="sqeuclidean")
>>> x, float(y)
([(0, 0), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5)], 1.117...)
```

Or by defining a custom distance function:

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

Or by using a precomputed distance matrix as input:

```
>>> from sklearn.metrics.pairwise import pairwise_distances
>>> dist_matrix = pairwise_distances(s1, s2, metric="sqeuclidean")
>>> x, y = dtw_path_from_metric(dist_matrix,
...                             metric="precomputed")
>>> x, float(y)
([(0, 0), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5)], 1.117...)
```

Examples using `tslearn.metrics.dtw_path_from_metric`

- *DTW computation with a custom distance metric*

3.7.14 dtw_accumulated_matrix

`tslearn.metrics.`**dtw_accumulated_matrix**(*s1*, *s2*, *mask*, *be=None*)

Compute the DTW accumulated cost matrix score between two time series.

It is not required that both time series share the same size, but they must be the same dimension.

Parameters

s1
[array-like, shape=(sz1,) or (sz1, d)] First time series.

s2
[array-like, shape=(sz2,) or (sz2, d)] Second time series.

mask
[array-like, shape=(sz1, sz2)] Mask used to constrain the region of computation. Unconsidered cells must have False values.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

mat
[array-like, shape=(sz1, sz2)] Accumulated cost matrix. Non computed cells due to masking have infinite value.

3.7.15 dtw_limited_warping_length

`tslearn.metrics.`**`dtw_limited_warping_length`**(*s1*, *s2*, *max_length*, *be=None*)

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

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

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

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

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

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

float
Similarity score

See also

dtw
Get the similarity score for DTW

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

References

[1], [2]

Examples

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

3.7.16 dtw_path_limited_warping_length

tslearn.metrics.**dtw_path_limited_warping_length**(*s1*, *s2*, *max_length*, *be=None*)

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

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

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

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

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

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

list of integer pairs
Optimal path

float
Similarity score

See also

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

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

References

[1], [2]

Examples

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

3.7.17 subsequence_path

tslearn.metrics.**subsequence_path**(*acc_cost_mat*, *idx_path_end*, *be=None*)

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

Parameters

acc_cost_mat: array-like, shape=(sz1, sz2)
Accumulated cost matrix comparing subsequence from a longer sequence.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

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

See also

dtw_subsequence_path
Get the similarity score for DTW

subsequence_cost_matrix
Calculate the required cost matrix

Examples

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

Examples using `tslearn.metrics.subsequence_path`

- *sDTW multi path matching*

3.7.18 subsequence_cost_matrix

`tslearn.metrics.`**`subsequence_cost_matrix`**(*subseq*, *longseq*, *be=None*)

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

Parameters

subseq
[array-like, shape=(sz1, d) or (sz1,)] Subsequence time series. If shape is (sz1,), the time series is assumed to be univariate.

longseq
[array-like, shape=(sz2, d) or (sz2,)] Reference time series. If shape is (sz2,), the time series is assumed to be univariate.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

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

Examples using `tslearn.metrics.subsequence_cost_matrix`

- *sDTW multi path matching*

3.7.19 dtw_subsequence_path

`tslearn.metrics.`**`dtw_subsequence_path`**(*subseq*, *longseq*, *be=None*)

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

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

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

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

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

Parameters

subseq
: [array-like, shape=(sz1, d) or (sz1,)] A query time series. If shape is (sz1,), the time series is assumed to be univariate.

longseq
: [array-like, shape=(sz2, d) or (sz2,)] A reference (supposed to be longer than *subseq*) time series. If shape is (sz2,), the time series is assumed to be univariate.

be
: [Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

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

float
: Similarity score

See also

dtw
: Get the similarity score for DTW

subsequence_cost_matrix
: Calculate the required cost matrix

subsequence_path
: Calculate a matching path manually

Examples

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

3.7.20 lcss

tslearn.metrics.**lcss**(*s1*, *s2*, *eps=1.0*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *be=None*)

Compute the Longest Common Subsequence (LCSS) similarity measure between (possibly multidimensional) time series and return the similarity.

LCSS is computed by matching indexes that are met up until the eps threshold, so it leaves some points unmatched and focuses on the similar parts of two sequences. The matching can occur even if the time indexes are different. One can set additional constraints to the set of acceptable paths: the Sakoe-Chiba band which is parametrized by a radius or the Itakura parallelogram which is parametrized by a maximum slope. Both these constraints consists in forcing paths to lie close to the diagonal. To retrieve a meaningful similarity value from the length of the longest common subsequence, the percentage of that value regarding the length of the shortest time series is returned.

According to this definition, the values returned by LCSS range from 0 to 1, the highest value taken when two time series fully match, and vice-versa. It is not required that both time series share the same size, but they must be the same dimension. LCSS was originally presented in [1] and is discussed in more details in our *dedicated user-guide page*.

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

eps
[float (default: 1.)] Maximum matching distance threshold.

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

sakoe_chiba_radius
[int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [1], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

itakura_max_slope
[float or None (default: None)] Maximum slope for the Itakura parallelogram constraint. If None and *global_constraint* is set to "itakura", a maximum slope of 2. is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which

constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

float
Similarity score

See also

lcss_path
Get both the matching path and the similarity score for LCSS

Notes

Contrary to Dynamic Time Warping and variants, an LCSS path does not need to be contiguous.

References

[1]

Examples

```
>>> lcss([1, 2, 3], [1., 2., 2., 3.])
1.0
>>> lcss([1, 2, 3], [1., 2., 2., 4., 7.])
1.0
>>> lcss([1, 2, 3], [1., 2., 2., 2., 3.], eps=0)
1.0
>>> lcss([1, 2, 3], [-2., 5., 7.], eps=3)
0.6666666666666666
```

3.7.21 lcss_path

tslearn.metrics.**lcss_path**(*s1*, *s2*, *eps=1*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *be=None*)

Compute the Longest Common Subsequence (LCSS) similarity measure between (possibly multidimensional) time series and return both the path and the similarity.

LCSS is computed by matching indexes that are met up until the eps threshold, so it leaves some points unmatched and focuses on the similar parts of two sequences. The matching can occur even if the time indexes are different. One can set additional constraints to the set of acceptable paths: the Sakoe-Chiba band which is parametrized by a radius or the Itakura parallelogram which is parametrized by a maximum slope. Both these constraints consists in forcing paths to lie close to the diagonal.

To retrieve a meaningful similarity value from the length of the longest common subsequence, the percentage of that value regarding the length of the shortest time series is returned.

According to this definition, the values returned by LCSS range from 0 to 1, the highest value taken when two time series fully match, and vice-versa. It is not required that both time series share the same size, but they must be the same dimension. LCSS was originally presented in [1] and is discussed in more details in our *dedicated user-guide page*.

Parameters

s1
: [array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
: [array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

eps
: [float (default: 1.)] Maximum matching distance threshold.

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

sakoe_chiba_radius
: [int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [1], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

be
: [Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

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

`lcss`
: Get only the similarity score for LCSS

lcss_path_from_metric
Compute LCSS using a user-defined distance metric

Notes

Contrary to Dynamic Time Warping and variants, an LCSS path does not need to be contiguous.

References

[1]

Examples

```
>>> path, sim = lcss_path([1., 2., 3.], [1., 2., 2., 3.])
>>> path
[(0, 1), (1, 2), (2, 3)]
>>> sim
1.0
>>> lcss_path([1., 2., 3.], [1., 2., 2., 4.])[1]
1.0
```

Examples using `tslearn.metrics.lcss_path`

- *Longest Common Subsequence*

3.7.22 lcss_path_from_metric

tslearn.metrics.**lcss_path_from_metric**(*s1*, *s2=None*, *eps=1*, *metric='euclidean'*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *be=None*, ***kwds*)

Compute the Longest Common Subsequence (LCSS) similarity measure between (possibly multidimensional) time series using a distance metric defined by the user and return both the path and the similarity.

Having the length of the longest commom subsequence between two time series, the similarity is computed as the percentage of that value regarding the length of the shortest time series.

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

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

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,) if metric!="precomputed", (sz1, sz2) otherwise] A time series or an array of pairwise distances between samples. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,), optional (default: None)] A second time series, only allowed if metric != "precomputed". If shape is (sz2,), the time series is assumed to be univariate.

eps
: [float (default: 1.)] Maximum matching distance threshold.

metric
: [string or callable (default: "euclidean")] Function used to compute the pairwise distances between each points of *s1* and *s2*. If metric is "precomputed", *s1* is assumed to be a distance matrix. If metric is an other string, it must be one of the options compatible with sklearn.metrics.pairwise_distances. Alternatively, if metric is a callable function, it is called on pairs of rows of *s1* and *s2*. The callable should take two 1 dimensional arrays as input and return a value indicating the distance between them.

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

sakoe_chiba_radius
: [int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [1], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

be
: [Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

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

Returns

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

float
: Similarity score.

See also

lcss
: Get only the similarity score for LCSS

lcss_path
: Get both the matching path and the similarity score for LCSS

Notes

By using a squared euclidean distance metric as shown above, the output path and similarity is the same as the one obtained by using lcss_path (which uses the euclidean distance) simply because with the sum of squared distances the matching threshold is still not reached. Also, contrary to Dynamic Time Warping and variants, an LCSS path does not need to be contiguous.

References

[1]

Examples

Lets create 2 numpy arrays to wrap:

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

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

```
>>> lcss_path_from_metric(s1, s2,
...                       metric="sqeuclidean")
([(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)], 1.0)
```

Or by defining a custom distance function:

```
>>> sqeuclidean = lambda x, y: np.sum((x-y)**2)
>>> lcss_path_from_metric(s1, s2, metric=sqeuclidean)
([(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)], 1.0)
```

Or by using a precomputed distance matrix as input:

```
>>> from sklearn.metrics.pairwise import pairwise_distances
>>> dist_matrix = pairwise_distances(s1, s2, metric="sqeuclidean")
>>> lcss_path_from_metric(dist_matrix,
...                       metric="precomputed")
([(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)], 1.0)
```

Examples using `tslearn.metrics.lcss_path_from_metric`

- *Longest Common Subsequence with a custom distance metric*

3.7.23 gak

`tslearn.metrics.`**gak**(*s1*, *s2*, *sigma=1.0*, *be=None*)

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

$$\mathrm{gak}(\mathbf{x}, \mathbf{y}) = \frac{k(\mathbf{x}, \mathbf{y})}{\sqrt{k(\mathbf{x}, \mathbf{x}) k(\mathbf{y}, \mathbf{y})}}$$

where

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

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 $gak(x, x) = 1$ for all x and $gak(x, y) \in [0, 1]$ for all x, y.

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

float
Kernel value

See also

cdist_gak
Compute cross-similarity matrix using Global Alignment kernel

References

[1]

Examples

```
>>> float(gak([1, 2, 3], [1., 2., 2., 3.], sigma=2.))
0.839...
>>> float(gak([1, 2, 3], [1., 2., 2., 3., 4.]))
0.273...
```

3.7.24 soft_dtw

`tslearn.metrics.soft_dtw(ts1, ts2, gamma=1.0, be=None, compute_with_backend=False)`

Compute Soft-DTW metric between two time series.

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

Soft-DTW is computed as:

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

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

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

Parameters

ts1
[array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

ts2
[array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

compute_with_backend
[bool, default=False] This parameter has no influence when the NumPy backend is used. When a backend different from NumPy is used (cf parameter *be*): If *True*, the computation is done with the corresponding backend. If *False*, a conversion to the NumPy backend can be used to accelerate the computation.

Returns

float
Similarity

See also

cdist_soft_dtw
Cross similarity matrix between time series datasets

References

[1]

Examples

```
>>> float(soft_dtw([1, 2, 2, 3],
...                [1., 2., 3., 4.],
...                gamma=1.))
-0.89...
>>> float(soft_dtw([1, 2, 3, 3],
...                [1., 2., 2.1, 3.2],
...                 gamma=0.01))
0.089...
```

The PyTorch backend can be used to compute gradients:

```
>>> import torch
>>> ts1 = torch.tensor([[1.0], [2.0], [3.0]], requires_grad=True)
>>> ts2 = torch.tensor([[3.0], [4.0], [-3.0]])
>>> sim = soft_dtw(ts1, ts2, gamma=1.0, be="pytorch", compute_with_backend=True)
>>> print(sim)
tensor(41.1876, dtype=torch.float64, grad_fn=<SelectBackward0>)
>>> sim.backward()
>>> print(ts1.grad)
tensor([[-4.0001],
        [-2.2852],
        [10.1643]])
```

```
>>> ts1_2d = torch.tensor([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]], requires_grad=True)
>>> ts2_2d = torch.tensor([[3.0, 3.0], [4.0, 4.0], [-3.0, -3.0]])
>>> sim = soft_dtw(ts1_2d, ts2_2d, gamma=1.0, be="pytorch", compute_with_
↪backend=True)
>>> print(sim)
tensor(83.2951, dtype=torch.float64, grad_fn=<SelectBackward0>)
>>> sim.backward()
>>> print(ts1_2d.grad)
tensor([[-4.0000, -4.0000],
        [-2.0261, -2.0261],
        [10.0206, 10.0206]])
```

3.7.25 soft_dtw_alignment

`tslearn.metrics.`**`soft_dtw_alignment`**(*ts1*, *ts2*, *gamma=1.0*, *be=None*, *compute_with_backend=False*)

Compute Soft-DTW metric between two time series and return both the similarity measure and the alignment matrix.

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

Soft-DTW is computed as:

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

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

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

Parameters

ts1
: [array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

ts2
: [array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

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

be
: [Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

compute_with_backend
: [bool, default=False] This parameter has no influence when the NumPy backend is used. When a backend different from NumPy is used (cf parameter *be*): If *True*, the computation is done with the corresponding backend. If *False*, a conversion to the NumPy backend can be used to accelerate the computation.

Returns

array-like, shape=(sz1, sz2)
: Soft-alignment matrix

float
: Similarity

See also

soft_dtw
: Returns soft-DTW score alone

References

[1]

Examples

```
>>> a, dist = soft_dtw_alignment([1, 2, 2, 3],
...                              [1., 2., 3., 4.],
...                              gamma=1.)
>>> float(dist)
-0.89...
>>> a
array([[1.00...e+00, 1.88...e-01, 2.83...e-04, 4.19...e-11],
       [3.40...e-01, 8.17...e-01, 8.87...e-02, 3.94...e-05],
       [5.05...e-02, 7.09...e-01, 5.30...e-01, 6.98...e-03],
       [1.37...e-04, 1.31...e-01, 7.30...e-01, 1.00...e+00]])
```

The PyTorch backend can be used to compute gradients:

```
>>> import torch
>>> ts1 = torch.tensor([[1.0], [2.0], [3.0]], requires_grad=True)
>>> ts2 = torch.tensor([[3.0], [4.0], [-3.0]])
>>> path, sim = soft_dtw_alignment(ts1, ts2, gamma=1.0, be="pytorch", compute_with_
↪backend=True)
>>> print(sim)
tensor(41.1876, dtype=torch.float64, grad_fn=<AsStridedBackward0>)
>>> sim.backward()
>>> print(ts1.grad)
tensor([[-4.0001],
```

(continues on next page)

(continued from previous page)

```
        [-2.2852],
        [10.1643]])
```

```
>>> ts1_2d = torch.tensor([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]], requires_grad=True)
>>> ts2_2d = torch.tensor([[3.0, 3.0], [4.0, 4.0], [-3.0, -3.0]])
>>> path, sim = soft_dtw_alignment(ts1_2d, ts2_2d, gamma=1.0, be="pytorch", compute_
↪with_backend=True)
>>> print(sim)
tensor(83.2951, dtype=torch.float64, grad_fn=<AsStridedBackward0>)
>>> sim.backward()
>>> print(ts1_2d.grad)
tensor([[-4.0000, -4.0000],
        [-2.0261, -2.0261],
        [10.0206, 10.0206]])
```

Examples using `tslearn.metrics.soft_dtw_alignment`

- *Soft Dynamic Time Warping*

3.7.26 lb_envelope

`tslearn.metrics.lb_envelope(ts, radius=1, be=None)`

Compute time series envelope as required by LB_Keogh.

LB_Keogh was originally presented in [1].

Parameters

ts
[array-like, shape=(sz, d) or (sz,)] Time series for which the envelope should be computed. If shape is (sz,), the time series is assumed to be univariate.

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).

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

envelope_down
[array-like, shape=(sz, d)] Lower-side of the envelope.

envelope_up
[array-like, shape=(sz, d)] Upper-side of the envelope.

See also

lb_keogh
Compute LB_Keogh similarity

References

[1]

Examples

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

Examples using `tslearn.metrics.lb_envelope`

- *LB_Keogh*

3.7.27 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, shape=(sz1, 1) or (sz1,)] Univariate query time series to compare to the envelope of the candidate.

ts_candidate
: [None or array-like, shape=(sz2, 1) or (sz2,) (default: None)] Univariate 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)
>>> float(lb_keogh(ts_query=ts2,
...                envelope_candidate=(env_low, env_up)))
2.8284...
>>> float(lb_keogh(ts_query=ts2,
...                ts_candidate=ts1,
...                radius=1))
2.8284...
```

Examples using `tslearn.metrics.lb_keogh`

- *LB_Keogh*

3.7.28 sigma_gak

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

Compute sigma value to be used for GAK.

This method was originally presented in [1].

Parameters

dataset
[array-like, shape=(n_ts, sz, d) or (n_ts, sz1) or (sz,)] A dataset of time series. If shape is (n_ts, sz), the dataset is composed of univariate time series. If shape is (sz,), the dataset is composed of a unique univariate 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.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

float
Suggested bandwidth (σ) for the Global Alignment kernel.

See also

gak
Compute Global Alignment kernel

cdist_gak
Compute cross-similarity matrix using Global Alignment kernel

References

[1]

Examples

```
>>> dataset = [[1, 2, 2, 3], [1., 2., 3., 4.]]
>>> float(sigma_gak(dataset=dataset,
...                 n_samples=200,
...                 random_state=0))
2.0...
```

3.7.29 gamma_soft_dtw

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

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

This method was originally presented in [1].

Parameters

dataset
[array-like, shape=(n_ts, sz, d) or (n_ts, sz1) or (sz,)] A dataset of time series. If shape is (n_ts, sz), the dataset is composed of univariate time series. If shape is (sz,), the dataset is composed of a unique univariate 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.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend*

or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

float
Suggested γ parameter for the Soft-DTW.

See also

sigma_gak
Compute sigma parameter for Global Alignment kernel

References

[1]

Examples

```
>>> dataset = [[1, 2, 2, 3], [1., 2., 3., 4.]]
>>> float(gamma_soft_dtw(dataset=dataset,
...                      n_samples=200,
...                      random_state=0))
8.0...
```

3.7.30 SoftDTWLossPyTorch

`tslearn.metrics.`**`SoftDTWLossPyTorch`**(*gamma=1.0*, *normalize=False*, *dist_func=None*)

Soft-DTW loss function in PyTorch.

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

Soft-DTW is computed as:

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

where d is a distance function or a dissimilarity measure supporting PyTorch automatic differentiation and $\min^\gamma$ is the soft-min operator of parameter γ defined as:

$$\min{}^\gamma\left(a_1, ..., a_n\right) = -\gamma \log \sum_{i=1}^{n} e^{-a_i/\gamma}$$

In the limit case $\gamma = 0$, $\min^\gamma$ reduces to a hard-min operator. The soft-DTW is then defined as the square of the DTW dissimilarity measure when d is the squared Euclidean distance.

Contrary to DTW, soft-DTW is not bounded below by zero, and we even have:

$$\text{soft-DTW}_\gamma(X, Y) \to -\infty \text{ when } \gamma \to +\infty$$

In [2], new dissimilarity measures are defined, that rely on soft-DTW. In particular, soft-DTW divergence is introduced to counteract the non-positivity of soft-DTW:

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

This divergence has the advantage of being minimized for $X = Y$ and being exactly 0 in that case.

Parameters

gamma
[float] Regularization parameter. It should be strictly positive. Lower is less smoothed (closer to true DTW).

normalize
[bool] If True, the Soft-DTW divergence is used. The Soft-DTW divergence is always positive. Optional, default: False.

dist_func
[callable] Distance function or dissimilarity measure. It takes two input arguments of shape (batch_size, ts_length, dim). It should support PyTorch automatic differentiation. Optional, default: None If None, the squared Euclidean distance is used.

See also

soft_dtw
Compute Soft-DTW metric between two time series.

cdist_soft_dtw
Compute cross-similarity matrix using Soft-DTW metric.

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

References

[1], [2]

Examples

```
>>> import torch
>>> from tslearn.metrics import SoftDTWLossPyTorch
>>> soft_dtw_loss = SoftDTWLossPyTorch(gamma=0.1)
>>> x = torch.zeros((4, 3, 2), requires_grad=True)
>>> y = torch.arange(0, 24).reshape(4, 3, 2)
>>> soft_dtw_loss_mean_value = soft_dtw_loss(x, y).mean()
>>> print(soft_dtw_loss_mean_value)
tensor(1081., grad_fn=<MeanBackward0>)
>>> soft_dtw_loss_mean_value.backward()
>>> print(x.grad.shape)
torch.Size([4, 3, 2])
>>> print(x.grad)
tensor([[[  0.0000,  -0.5000],
         [ -1.0000,  -1.5000],
         [ -2.0000,  -2.5000]],

        [[ -3.0000,  -3.5000],
         [ -4.0000,  -4.5000],
         [ -5.0000,  -5.5000]],

        [[ -6.0000,  -6.5000],
```

(continues on next page)

(continued from previous page)

```
         [ -7.0000,  -7.5000],
         [ -8.0000,  -8.5000]],

        [[ -9.0000,  -9.5000],
         [-10.0000, -10.5000],
         [-11.0000, -11.5000]]])
```

Examples using `tslearn.metrics.SoftDTWLossPyTorch`

- *Soft-DTW loss for PyTorch neural network*

3.7.31 frechet

`tslearn.metrics.`**`frechet`**(*s1*, *s2*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *be=None*)

Compute Frechet similarity [1] measure between (possibly multidimensional) time series and return it.

Frechet similarity score is computed as the maximum distance between aligned time series, i.e., if π is an optimal alignment path:

$$Frechet(X, Y) = \max_{(i,j)\in\pi} \|X_i - Y_j\|$$

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.

Parameters

s1
: [array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
: [array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

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

sakoe_chiba_radius
: [int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [2], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

be
: [Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

float
: Similarity score

See also

frechet_path
: Get both the matching path and the similarity score

frechet_path_from_metric
: Compute similarity score and path using a user-defined distance metric

cdist_frechet
: Cross similarity matrix between time series datasets

References

[1], [2]

Examples

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

The PyTorch backend can be used to compute gradients:

```
>>> import torch
>>> s1 = torch.tensor([[1.0], [2.0], [3.0]], requires_grad=True)
>>> s2 = torch.tensor([[3.0], [4.0], [-3.0]])
>>> sim = frechet(s1, s2, be="pytorch")
>>> print(sim)
tensor(6., dtype=torch.float64, grad_fn=<SqrtBackward0>)
>>> sim.backward()
>>> print(s1.grad)
tensor([[0.],
        [0.],
        [1.]])
```

```
>>> s1_2d = torch.tensor([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]], requires_grad=True)
>>> s2_2d = torch.tensor([[3.0, 3.0], [4.0, 4.0], [-3.0, -3.0]])
>>> sim = frechet(s1_2d, s2_2d, be="pytorch")
>>> print(sim)
tensor(8.4853, dtype=torch.float64, grad_fn=<SqrtBackward0>)
>>> sim.backward()
>>> print(s1_2d.grad)
```

(continues on next page)

(continued from previous page)

```
tensor([[0.0000, 0.0000],
        [0.0000, 0.0000],
        [0.7071, 0.7071]])
```

3.7.32 frechet_path

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

Compute Frechet similarity measure [1] between (possibly multidimensional) time series and an optimal alignment path.

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

$$Frechet(X, Y) = \max_{(i,j)\in\pi} \|X_i - Y_j\|$$

It is not required that both time series share the same size, but they must be the same dimension.

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,)] A time series. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,)] Another time series. If shape is (sz2,), the time series is assumed to be univariate.

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

sakoe_chiba_radius
[int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [2], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

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

frechet
Get only the similarity score

frechet_path_from_metric
Compute similarity score and path using a user-defined distance metric

cdist_frechet
Cross similarity matrix between time series datasets

References

[1], [2]

Examples

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

Examples using `tslearn.metrics.frechet_path`

- *Frechet*

3.7.33 frechet_path_from_metric

`tslearn.metrics.`**`frechet_path_from_metric`**(*s1*, *s2=None*, *metric='precomputed'*, *global_constraint=None*, *sakoe_chiba_radius=None*, *itakura_max_slope=None*, *be=None*, ***kwds*)

Compute Frechet similarity measure and an optimal alignment path [1] between (possibly multidimensional) time series using a distance metric defined by the user.

It is not required that both time series share the same size, but they must be the same dimension.

When using Pytorch backend only “precomputed”, “euclidean”, “sqeuclidean” and callable metrics are available. Otherwise, valid values for metric are the same as for scikit-learn pairwise_distances function i.e. a string (e.g. “euclidean”, “sqeuclidean”, “hamming”) or a function that is used to compute the pairwise distances. See scikit and scipy documentations for more information about the available metrics.

Parameters

s1
[array-like, shape=(sz1, d) or (sz1,) if metric!=”precomputed”, (sz1, sz2) otherwise] A time

series or an array of pairwise distances between samples. If shape is (sz1,), the time series is assumed to be univariate.

s2
[array-like, shape=(sz2, d) or (sz2,), optional (default: None)] A second time series, only used if metric != "precomputed". If shape is (sz2,), the time series is assumed to be univariate.

metric
[string or callable (default: "precomputed")] If metric is "precomputed", *s1* is assumed to be a distance matrix.

Otherwise, function used to compute the pairwise distances between each points of *s1* and *s2*. If metric is a string, it must be one of the options compatible with sklearn.metrics.pairwise_distances. Alternatively, if metric is a callable function, it is called on pairs of rows of *s1* and *s2*. The callable should take two 1 dimensional arrays as input and return a value indicating the distance between them.

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

sakoe_chiba_radius
[int or None (default: None)] Radius to be used for Sakoe-Chiba band global constraint. The Sakoe-Chiba radius corresponds to the parameter δ mentioned in [2], it controls how far in time we can go in order to match a given point from one time series to a point in another time series. If None and *global_constraint* is set to "sakoe_chiba", a radius of 1 is used. If both *sakoe_chiba_radius* and *itakura_max_slope* are set, *global_constraint* is used to infer which constraint to use among the two. In this case, if *global_constraint* corresponds to no global constraint, a *RuntimeWarning* is raised and no global constraint is used.

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

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

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

Returns

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

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

See also

frechet
: Get only the similarity score

frechet_path
: Get both the matching path and the similarity score

cdist_frechet
: Cross similarity matrix between time series datasets

Notes

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

References

[1], [2]

Examples

Lets create 2 numpy arrays to wrap:

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

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

```
>>> x, y = frechet_path_from_metric(s1, s2,
...                                 metric="sqeuclidean")
>>> x, float(y)
([(0, 0), (1, 0), (2, 1), (2, 2), (2, 3), (3, 4), (4, 5)], 0.4365...)
```

Or by defining a custom distance function:

```
>>> sqeuclidean = lambda x, y: np.sum((x-y)**2)
>>> x, y = frechet_path_from_metric(s1, s2, metric=sqeuclidean)
>>> x, float(y)
([(0, 0), (1, 0), (2, 1), (2, 2), (2, 3), (3, 4), (4, 5)], 0.4365...)
```

Or by using a precomputed distance matrix as input:

```
>>> from sklearn.metrics.pairwise import pairwise_distances
>>> dist_matrix = pairwise_distances(s1, s2, metric="sqeuclidean")
>>> x, y = frechet_path_from_metric(dist_matrix,
...                                 metric="precomputed")
>>> x, float(y)
([(0, 0), (1, 0), (2, 1), (2, 2), (2, 3), (3, 4), (4, 5)], 0.4365...)
```

3.7.34 frechet_accumulated_matrix

tslearn.metrics.**frechet_accumulated_matrix**(*s1*, *s2*, *mask*, *be=None*)

Compute the Frechet accumulated cost matrix score between two time series.

It is not required that both time series share the same size, but they must be the same dimension.

Parameters

s1
[array-like, shape=(sz1,) or (sz1, d)] First time series.

s2
[array-like, shape=(sz2,) or (sz2, d)] Second time series.

mask
[array-like, shape=(sz1, sz2)] Mask used to constrain the region of computation. Unconsidered cells must have False values.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

mat
[array-like, shape=(sz1, sz2)] Accumulated cost matrix. Non computed cells due to masking have infinite value.

3.8 tslearn.neural_network

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

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

Classes

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

3.8.1 TimeSeriesMLPClassifier

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

A Multi-Layer Perceptron classifier for time series.

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

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

Notes

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

Examples

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

Methods

fit(X, y)	Fit the model using X as training data and y as target values
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
partial_fit(X, y, *args, **kwargs)	Update the model with a single iteration over the given data.
predict(X)	Predict the class labels for the provided data
predict_log_proba(X)	Return the log of probability estimates.
predict_proba(X)	Predict the class probabilities for the provided data
score(X, y[, sample_weight])	Return accuracy on provided data and labels.
set_fit_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `fit` method.
set_params(**params)	Set the parameters of this estimator.
set_partial_fit_request(*[, classes, ...])	Configure whether metadata should be requested to be passed to the `partial_fit` method.
set_score_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `score` method.

fit(*X*, *y*)

Fit the model using X as training data and y as target values

Parameters

X

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

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

Returns

TimeSeriesMLPClassifier
The fitted estimator

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`partial_fit(X, y, *args, **kwargs)`

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

Parameters

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

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

Target values.
***args, ****kwargs**
[arguments for the underlying]

MLPClassifier's method from scikit-learn

Returns

TimeSeriesMLPClassifier
The fitted estimator

`predict(X)`

Predict the class labels for the provided data

Parameters

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

Returns

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

`predict_log_proba`(*X*)

Return the log of probability estimates.

Parameters

X
[ndarray of shape (n_samples, n_features)] The input data.

Returns

log_y_prob
[ndarray of shape (n_samples, n_classes)] The predicted log-probability of the sample for each class in the model, where classes are ordered as they are in *self.classes_*. Equivalent to *log(predict_proba(X))*.

`predict_proba`(*X*)

Predict the class probabilities for the provided data

Parameters

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

Returns

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

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

Return accuracy on provided data and labels.

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

Parameters

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

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

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

Returns

score
[float] Mean accuracy of `self.predict(X)` w.r.t. *y*.

`set_fit_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesMLPClassifier*

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.

- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `fit`.

Returns

self
[object] The updated object.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`set_partial_fit_request`(*, *classes: bool | None | str = '$UNCHANGED$'*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesMLPClassifier*

Configure whether metadata should be requested to be passed to the `partial_fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

classes
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `partial_fit`.

Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesMLPClassifier*

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

Returns

self
[object] The updated object.

3.8.2 TimeSeriesMLPRegressor

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

A Multi-Layer Perceptron regressor for time series.

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

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

Notes

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

Examples

```
>>> mlp = TimeSeriesMLPRegressor(hidden_layer_sizes=(64, 64),
...                              random_state=0)
>>> mlp.fit(X=[[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0, 0, 1])
TimeSeriesMLPRegressor(...)
>>> [c.shape for c in mlp.coefs_]
[(3, 64), (64, 64), (64, 1)]
>>> [c.shape for c in mlp.intercepts_]
[(64,), (64,), (1,)]
```

Methods

fit(X, y)	Fit the model using X as training data and y as target values
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
partial_fit(X, y, *args, **kwargs)	Update the model with a single iteration over the given data.
predict(X)	Predict the target for the provided data
score(X, y[, sample_weight])	Return coefficient of determination on test data.
set_fit_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `fit` method.
set_params(**params)	Set the parameters of this estimator.
set_partial_fit_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `partial_fit` method.
set_score_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `score` method.

`fit`(*X*, *y*)

Fit the model using X as training data and y as target values

Parameters

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

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

Returns

TimeSeriesMLPRegressor
The fitted estimator

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`partial_fit`(*X*, *y*, **args*, ***kwargs*)

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

Parameters

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

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

Target values.
***args, **kwargs**
[arguments for the underlying]

MLPClassifier's method from scikit-learn

Returns

TimeSeriesMLPRegressor
The fitted estimator

`predict`(*X*)

Predict the target for the provided data

Parameters

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

Returns

array, shape = (n_ts,) or (n_ts, dim_y)
Array of predicted targets

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

Return coefficient of determination on test data.

The coefficient of determination, R^2, is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares `((y_true - y_pred)** 2).sum()` and v is the total sum of squares `((y_true - y_true.mean()) ** 2).sum()`. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.

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

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

Returns

score
[float] R^2 of `self.predict(X)` w.r.t. *y*.

Notes

The R^2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

`set_fit_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesMLPRegressor*

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.

- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `fit`.

Returns

self
[object] The updated object.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`set_partial_fit_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesMLPRegressor*

Configure whether metadata should be requested to be passed to the `partial_fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `partial_fit`.

Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesMLPRegressor*

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

Returns

self
[object] The updated object.

3.9 tslearn.neighbors

The *tslearn.neighbors* module gathers nearest neighbor algorithms using time series metrics.

Classes

KNeighborsTimeSeries([n_neighbors, metric, ...])	Unsupervised learner for implementing neighbor searches for Time Series.
KNeighborsTimeSeriesClassifier([...])	Classifier implementing the k-nearest neighbors vote for Time Series.
KNeighborsTimeSeriesRegressor([n_neighbors, ...])	Regressor implementing the k-nearest neighbors vote for Time Series.

3.9.1 KNeighborsTimeSeries

class tslearn.neighbors.**KNeighborsTimeSeries**(*n_neighbors=5*, *metric='dtw'*, *metric_params=None*, *n_jobs=None*, *verbose=0*)

Unsupervised learner for implementing neighbor searches for Time Series.

Parameters

n_neighbors
[int (default: 5)] Number of nearest neighbors to be considered for the decision.

metric
[{'dtw', 'softdtw', 'ctw', 'euclidean', 'sqeuclidean', 'cityblock', 'sax'} (default: 'dtw')] Metric to be used at the core of the nearest neighbor procedure. DTW and SAX are described in more detail in *tslearn.metrics*. When SAX is provided as a metric, the data is expected to be normalized such that each time series has zero mean and unit variance. Other metrics are described in scipy.spatial.distance doc.

metric_params
[dict or None (default: None)] Dictionary of metric parameters. For metrics that accept parallelization of the cross-distance matrix computations, *n_jobs* and *verbose* keys passed in *metric_params* are overridden by the *n_jobs* and *verbose* arguments. For 'sax' metric, these are hyper-parameters to be passed at the creation of the *SymbolicAggregateApproximation* object.

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

Notes

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

Examples

```
>>> time_series = to_time_series_dataset([[1, 2, 3, 4],
...                                       [3, 3, 2, 0],
...                                       [1, 2, 2, 4]])
>>> knn = KNeighborsTimeSeries(n_neighbors=1).fit(time_series)
>>> dataset = to_time_series_dataset([[1, 1, 2, 2, 2, 3, 4]])
>>> dist, ind = knn.kneighbors(dataset, return_distance=True)
>>> dist
array([[0.]])
>>> print(ind)
[[0]]
>>> knn2 = KNeighborsTimeSeries(n_neighbors=10,
...                             metric="euclidean").fit(time_series)
>>> print(knn2.kneighbors(return_distance=False))
[[2 1]
 [2 0]
 [0 1]]
```

Methods

fit(X[, y])	Fit the model using X as training data
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors_graph([X, n_neighbors, mode])	Compute the (weighted) graph of k-Neighbors for points in X.
radius_neighbors([X, radius, ...])	Find the neighbors within a given radius of a point or points.
radius_neighbors_graph([X, radius, mode, ...])	Compute the (weighted) graph of Neighbors for points in X.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.

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

Fit the model using X as training data

Parameters

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

classmethod from_hdf5(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`kneighbors`(*X=None*, *n_neighbors=None*, *return_distance=True*)

Finds the K-neighbors of a point.

Returns indices of and distances to the neighbors of each point.

Parameters

X
[array-like, shape (n_ts, sz, d)] The query time series. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors
[int] Number of neighbors to get (default is the value passed to the constructor).

return_distance
[boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist
[array] Array representing the distance to points, only present if return_distance=True

ind
[array] Indices of the nearest points in the population matrix.

`kneighbors_graph`(*X=None*, *n_neighbors=None*, *mode='connectivity'*)

Compute the (weighted) graph of k-Neighbors for points in X.

Parameters

X
[{array-like, sparse matrix} of shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed', default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor. For `metric='precomputed'` the shape should be (n_queries, n_indexed). Otherwise the shape should be (n_queries, n_features).

n_neighbors
[int, default=None] Number of neighbors for each sample. The default is the value passed to the constructor.

mode
[{'connectivity', 'distance'}, default='connectivity'] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are distances between points, type of distance depends on the selected metric parameter in NearestNeighbors class.

Returns

A
[sparse-matrix of shape (n_queries, n_samples_fit)] *n_samples_fit* is the number of samples in the fitted data. *A[i, j]* gives the weight of the edge connecting *i* to *j*. The matrix is of CSR format.

See also

`NearestNeighbors.radius_neighbors_graph`
Compute the (weighted) graph of Neighbors for points in X.

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

`radius_neighbors`(*X=None*, *radius=None*, *return_distance=True*, *sort_results=False*)

Find 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, sparse matrix} of (n_samples, n_features), default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius
[float, default=None] Limiting distance of neighbors to return. The default is the value passed to the constructor.

return_distance
[bool, default=True] Whether or not to return the distances.

sort_results
[bool, default=False] If True, the distances and indices will be sorted by increasing distances before being returned. If False, the results may not be sorted. If *return_distance=False*, setting *sort_results=True* will result in an error.

Added in version 0.22.

Returns

neigh_dist
[ndarray of shape (n_samples,) of arrays] Array representing the distances to each point, only present if *return_distance=True*. The distance values are computed according to the `metric` constructor parameter.

neigh_ind
[ndarray of shape (n_samples,) of arrays] An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, *radius_neighbors* returns arrays of objects, where each object is a 1D array of indices or distances.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1, 1, 1]:

```
>>> import numpy as np
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.6)
>>> neigh.fit(samples)
NearestNeighbors(radius=1.6)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0]))
[1.5 0.5]
>>> print(np.asarray(rng[1][0]))
[1 2]
```

The first array returned contains the distances to all points which are closer than 1.6, while the second array returned contains their indices. In general, multiple points can be queried at the same time.

radius_neighbors_graph(*X=None*, *radius=None*, *mode='connectivity'*, *sort_results=False*)

Compute the (weighted) graph of Neighbors for points in X.

Neighborhoods are restricted the points at a distance lower than radius.

Parameters

X
[{array-like, sparse matrix} of shape (n_samples, n_features), default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius
[float, default=None] Radius of neighborhoods. The default is the value passed to the constructor.

mode
[{'connectivity', 'distance'}, default='connectivity'] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are distances between points, type of distance depends on the selected metric parameter in NearestNeighbors class.

sort_results
[bool, default=False] If True, in each row of the result, the non-zero entries will be sorted by increasing distances. If False, the non-zero entries may not be sorted. Only used with mode='distance'.

Added in version 0.22.

Returns

A
[sparse-matrix of shape (n_queries, n_samples_fit)] *n_samples_fit* is the number of samples in the fitted data. *A[i, j]* gives the weight of the edge connecting *i* to *j*. The matrix is of CSR format.

See also

kneighbors_graph
Compute the (weighted) graph of k-Neighbors for points in X.

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(radius=1.5)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`to_hdf5`(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

`to_json`(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

`to_pickle`(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

Examples using `tslearn.neighbors.KNeighborsTimeSeries`

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

3.9.2 KNeighborsTimeSeriesClassifier

`class tslearn.neighbors.KNeighborsTimeSeriesClassifier`(*n_neighbors=5*, *weights='uniform'*, *metric='dtw'*, *metric_params=None*, *n_jobs=None*, *verbose=0*)

Classifier implementing the k-nearest neighbors vote for Time Series.

Parameters

n_neighbors
[int (default: 5)] Number of nearest neighbors to be considered for the decision.

weights
[str or callable, optional (default: 'uniform')] Weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

metric
: [one of the metrics allowed for *KNeighborsTimeSeries* class (default: 'dtw')] Metric to be used at the core of the nearest neighbor procedure

metric_params
: [dict or None (default: None)] Dictionnary of metric parameters. For metrics that accept parallelization of the cross-distance matrix computations, *n_jobs* and *verbose* keys passed in *metric_params* are overridden by the *n_jobs* and *verbose* arguments. For 'sax' metric, these are hyper-parameters to be passed at the creation of the *SymbolicAggregateApproximation* object.

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

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

Notes

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

Examples

```
>>> clf = KNeighborsTimeSeriesClassifier(n_neighbors=2, metric="dtw")
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0, 0, 1]).predict([[1, 2.2, 3.5]])
array([0])
>>> clf = KNeighborsTimeSeriesClassifier(n_neighbors=2,
...                                      metric="dtw",
...                                      n_jobs=2)
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0, 0, 1]).predict([[1, 2.2, 3.5]])
array([0])
>>> clf = KNeighborsTimeSeriesClassifier(n_neighbors=2,
...                                      metric="dtw",
...                                      metric_params={
...                                          "itakura_max_slope": 2.},
...                                      n_jobs=2)
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0, 0, 1]).predict([[1, 2.2, 3.5]])
array([0])
```

Methods

fit(X, y)	Fit the model using X as training data and y as target values
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.

continues on next page

Table 23 – continued from previous page

from_pickle(path)	Load model from a pickle file.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors_graph([X, n_neighbors, mode])	Compute the (weighted) graph of k-Neighbors for points in X.
predict(X)	Predict the class labels for the provided data
predict_proba(X)	Predict the class probabilities for the provided data
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
set_score_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `score` method.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.

fit(*X*, *y*)

Fit the model using X as training data and y as target values

Parameters

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

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

Returns

KNeighborsTimeSeriesClassifier
The fitted estimator

classmethod from_hdf5(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`kneighbors(X=None, n_neighbors=None, return_distance=True)`

Finds the K-neighbors of a point.

Returns indices of and distances to the neighbors of each point.

Parameters

X
[array-like, shape (n_ts, sz, d)] The query time series. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors
[int] Number of neighbors to get (default is the value passed to the constructor).

return_distance
[boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist
[array] Array representing the distance to points, only present if return_distance=True

ind
[array] Indices of the nearest points in the population matrix.

`kneighbors_graph(X=None, n_neighbors=None, mode='connectivity')`

Compute the (weighted) graph of k-Neighbors for points in X.

Parameters

X
[{array-like, sparse matrix} of shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed', default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor. For `metric='precomputed'` the shape should be (n_queries, n_indexed). Otherwise the shape should be (n_queries, n_features).

n_neighbors
[int, default=None] Number of neighbors for each sample. The default is the value passed to the constructor.

mode
[{'connectivity', 'distance'}, default='connectivity'] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are distances between points, type of distance depends on the selected metric parameter in NearestNeighbors class.

Returns

A
[sparse-matrix of shape (n_queries, n_samples_fit)] *n_samples_fit* is the number of samples in the fitted data. *A[i, j]* gives the weight of the edge connecting *i* to *j*. The matrix is of CSR format.

See also

`NearestNeighbors.radius_neighbors_graph`
Compute the (weighted) graph of Neighbors for points in X.

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

`predict`(*X*)

Predict the class labels for the provided data

Parameters

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

Returns

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

`predict_proba`(*X*)

Predict the class probabilities for the provided data

Parameters

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

Returns

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

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

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

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

Parameters

X
[array-like of shape (n_samples, n_features), or None] Test samples. If *None*, predictions for all indexed points are used; in this case, points are not considered their own neighbors. This means that *knn.fit(X, y).score(None, y)* implicitly performs a leave-one-out cross-validation procedure and is equivalent to *cross_val_score(knn, X, y, cv=LeaveOneOut())* but typically much faster.

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

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

Returns

score
[float] Mean accuracy of `self.predict(X)` w.r.t. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *KNeighborsTimeSeriesClassifier*

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

Returns

self
[object] The updated object.

`to_hdf5`(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

`to_json`(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

`to_pickle`(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

Examples using `tslearn.neighbors.KNeighborsTimeSeriesClassifier`

- *Hyper-parameter tuning of a pipeline with KNeighbors time series classifier*
- *Nearest neighbors*
- *1-NN with SAX + MINDIST*

3.9.3 KNeighborsTimeSeriesRegressor

class tslearn.neighbors.**KNeighborsTimeSeriesRegressor**(*n_neighbors=5*, *weights='uniform'*, *metric='dtw'*, *metric_params=None*, *n_jobs=None*, *verbose=0*)

Regressor implementing the k-nearest neighbors vote for Time Series.

Parameters

n_neighbors
[int (default: 5)] Number of nearest neighbors to be considered for the decision.

weights
[str or callable, optional (default: 'uniform')] Weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

metric
[one of the metrics allowed for *KNeighborsTimeSeries* class (default: 'dtw')] Metric to be used at the core of the nearest neighbor procedure

metric_params
[dict or None (default: None)] Dictionnary of metric parameters. For metrics that accept parallelization of the cross-distance matrix computations, *n_jobs* and *verbose* keys passed in *metric_params* are overridden by the *n_jobs* and *verbose* arguments. For 'sax' metric, these are hyper-parameters to be passed at the creation of the *SymbolicAggregateApproximation* object.

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

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

Examples

```
>>> clf = KNeighborsTimeSeriesRegressor(n_neighbors=2, metric="dtw")
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0.1, 0.1, 1.1]).predict([[1, 2.2, 3.5]])
array([0.1])
>>> clf = KNeighborsTimeSeriesRegressor(n_neighbors=2,
...                                     metric="dtw",
...                                     n_jobs=2)
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0.1, 0.1, 1.1]).predict([[1, 2.2, 3.5]])
array([0.1])
```

(continues on next page)

(continued from previous page)

```
>>> clf = KNeighborsTimeSeriesRegressor(n_neighbors=2,
...                                     metric="dtw",
...                                     metric_params={
...                                         "itakura_max_slope": 2.},
...                                     n_jobs=2)
>>> clf.fit([[1, 2, 3], [1, 1.2, 3.2], [3, 2, 1]],
...         y=[0.1, 0.1, 1.1]).predict([[1, 2.2, 3.5]])
array([0.1])
```

Methods

fit(X, y)	Fit the model using X as training data and y as target values
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors_graph([X, n_neighbors, mode])	Compute the (weighted) graph of k-Neighbors for points in X.
predict(X)	Predict the target for the provided data
score(X, y[, sample_weight])	Return coefficient of determination on test data.
set_params(**params)	Set the parameters of this estimator.
set_score_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `score` method.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.

fit(*X*, *y*)

Fit the model using X as training data and y as target values

Parameters

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

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

Returns

KNeighborsTimeSeriesRegressor
The fitted estimator

classmethod from_hdf5(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_json`(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_pickle`(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`kneighbors`(*X=None*, *n_neighbors=None*, *return_distance=True*)

Finds the K-neighbors of a point.

Returns indices of and distances to the neighbors of each point.

Parameters

X
[array-like, shape (n_ts, sz, d)] The query time series. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors
[int] Number of neighbors to get (default is the value passed to the constructor).

return_distance
[boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist
[array] Array representing the distance to points, only present if return_distance=True

ind
[array] Indices of the nearest points in the population matrix.

kneighbors_graph(*X=None*, *n_neighbors=None*, *mode='connectivity'*)

Compute the (weighted) graph of k-Neighbors for points in X.

Parameters

X
[{array-like, sparse matrix} of shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed', default=None] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor. For `metric='precomputed'` the shape should be (n_queries, n_indexed). Otherwise the shape should be (n_queries, n_features).

n_neighbors
[int, default=None] Number of neighbors for each sample. The default is the value passed to the constructor.

mode
[{'connectivity', 'distance'}, default='connectivity'] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are distances between points, type of distance depends on the selected metric parameter in NearestNeighbors class.

Returns

A
[sparse-matrix of shape (n_queries, n_samples_fit)] *n_samples_fit* is the number of samples in the fitted data. *A[i, j]* gives the weight of the edge connecting *i* to *j*. The matrix is of CSR format.

See also

NearestNeighbors.radius_neighbors_graph
Compute the (weighted) graph of Neighbors for points in X.

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

`predict`(*X*)

Predict the target for the provided data

Parameters

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

Returns

array, shape = (n_ts,) or (n_ts, dim_y)
Array of predicted targets

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

Return coefficient of determination on test data.

The coefficient of determination, R^2, is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares `((y_true - y_pred)** 2).sum()` and v is the total sum of squares `((y_true - y_true.mean()) ** 2).sum()`. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.

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

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

Returns

score
[float] R^2 of `self.predict(X)` w.r.t. *y*.

Notes

The R^2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *KNeighborsTimeSeriesRegressor*

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

Returns

self
[object] The updated object.

to_hdf5(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

to_json(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

to_pickle(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

3.10 tslearn.piecewise

The *tslearn.piecewise* module gathers time series piecewise approximation algorithms.

Classes

OneD_SymbolicAggregateApproximation([...])	One-D Symbolic Aggregate approXimation (1d-SAX) transformation.
PiecewiseAggregateApproximation([n_segments])	Piecewise Aggregate Approximation (PAA) transformation.
SymbolicAggregateApproximation([n_segments, ...])	Symbolic Aggregate approXimation (SAX) transformation.

3.10.1 OneD_SymbolicAggregateApproximation

class tslearn.piecewise.**OneD_SymbolicAggregateApproximation**(*n_segments=1*, *alphabet_size_avg=5*, *alphabet_size_slope=5*, *sigma_l=None*, *scale=False*)

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

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

Parameters

n_segments
[int (default: 1)] Number of PAA segments to compute.

alphabet_size_avg
[int (default: 5)] Number of SAX symbols to use to describe average values.

alphabet_size_slope
[int (default: 5)] Number of SAX symbols to use to describe slopes.

sigma_l
[float or None (default: None)] Scale parameter of the Gaussian distribution used to quantize slopes. If None, the formula given in [1] is used: $\sigma_L = \sqrt{0.03/L}$ where L is the length of each segment.

scale: bool (default: False)
Whether input data should be scaled for each feature of each time series to have zero mean and unit variance. Default for this parameter is set to *False* in version 0.4 to ensure backward compatibility, but is likely to change in a future version.

Attributes

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

breakpoints_slope_
[numpy.ndarray of shape (alphabet_size_slope - 1,)] List of breakpoints used to generate SAX symbols for slopes.

Notes

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

References

[1]

Examples

```
>>> one_d_sax = OneD_SymbolicAggregateApproximation(n_segments=3,
...         alphabet_size_avg=2, alphabet_size_slope=2, sigma_l=1.)
>>> data = [[-1., 2., 0.1, -1., 1., -1.], [1., 3.2, -1., -3., 1., -1.]]
>>> one_d_sax_data = one_d_sax.fit_transform(data)
>>> one_d_sax_data.shape
(2, 3, 2)
>>> one_d_sax_data
array([[[1, 1],
        [0, 0],
        [1, 0]],

       [[1, 1],
        [0, 0],
        [1, 0]]])
>>> one_d_sax.distance_sax(one_d_sax_data[0], one_d_sax_data[1])
0.0
>>> one_d_sax.distance(data[0], data[1])
0.0
>>> one_d_sax.inverse_transform(one_d_sax_data)
array([[[ 0.33724488],
        [ 1.01173463],
        [-0.33724488],
        [-1.01173463],
        [ 1.01173463],
        [ 0.33724488]],

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

Methods

distance(ts1, ts2)	Compute distance between 1d-SAX representations as defined in [1].
distance_1d_sax(sax1, sax2)	Compute distance between 1d-SAX representations as defined in [1].
distance_paa(paa1, paa2)	Compute distance between PAA representations as defined in [1].

continues on next page

Table 26 – continued from previous page

distance_sax(sax1, sax2)	Compute distance between SAX representations as defined in [1].
fit(X[, y])	Fit a 1d-SAX representation.
fit_transform(X[, y])	Fit a 1d-SAX representation and transform the data accordingly.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Compute time series corresponding to given 1d-SAX representations.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X[, y])	Transform a dataset of time series into its 1d-SAX representation.

distance(*ts1*, *ts2*)

Compute distance between 1d-SAX representations as defined in [1].

Parameters

ts1
[array-like] A time series

ts2
[array-like] Another time series

Returns

float
1d-SAX distance

References

[1]

distance_1d_sax(*sax1*, *sax2*)

Compute distance between 1d-SAX representations as defined in [1].

Parameters

sax1
[array-like] 1d-SAX representation of a time series

sax2
[array-like] 1d-SAX representation of another time series

Returns

float
1d-SAX distance

Notes

Unlike SAX distance, 1d-SAX distance does not lower bound Euclidean distance between original time series.

References

[1]

`distance_paa`(*paa1*, *paa2*)

Compute distance between PAA representations as defined in [1].

Parameters

paa1
[array-like] PAA representation of a time series

paa2
[array-like] PAA representation of another time series

Returns

float
PAA distance

References

[1]

`distance_sax`(*sax1*, *sax2*)

Compute distance between SAX representations as defined in [1].

Parameters

sax1
[array-like] SAX representation of a time series

sax2
[array-like] SAX representation of another time series

Returns

float
SAX distance

References

[1]

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

Fit a 1d-SAX representation.

Parameters

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

Returns

OneD_SymbolicAggregateApproximation
self

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

Fit a 1d-SAX representation and transform the data accordingly.

Parameters

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

Returns

numpy.ndarray of integers with shape (n_ts, n_segments, 2 * d)
1d-SAX-Transformed dataset. The order of the last dimension is: first d elements represent average values (standard SAX symbols) and the last d are for slopes

`classmethod from_hdf5`(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_json`(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_pickle`(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Compute time series corresponding to given 1d-SAX representations.

Parameters

X
[array-like of shape (n_ts, sz_sax, 2 * d)] A dataset of SAX series.

Returns

numpy.ndarray of shape (n_ts, sz_original_ts, d)
A dataset of time series corresponding to the provided representation.

`set_output`(*, *transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params`(**params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`to_hdf5`(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

`to_json`(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

`to_pickle`(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

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

Transform a dataset of time series into its 1d-SAX representation.

Parameters

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

Returns

numpy.ndarray of integers with shape (n_ts, n_segments, 2 * d)
1d-SAX-Transformed dataset

Examples using `tslearn.piecewise.OneD_SymbolicAggregateApproximation`

- *PAA and SAX features*

3.10.2 PiecewiseAggregateApproximation

`class tslearn.piecewise.PiecewiseAggregateApproximation`(*n_segments=1*)

Piecewise Aggregate Approximation (PAA) transformation.

PAA was originally presented in [1].

Parameters

n_segments
[int (default: 1)] Number of PAA segments to compute

Notes

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

References

[1]

Examples

```
>>> paa = PiecewiseAggregateApproximation(n_segments=3)
>>> data = [[-1., 2., 0.1, -1., 1., -1.], [1., 3.2, -1., -3., 1., -1.]]
>>> paa_data = paa.fit_transform(data)
>>> paa_data.shape
(2, 3, 1)
>>> paa_data
array([[[ 0.5 ],
        [-0.45],
        [ 0.  ]],

       [[ 2.1 ],
        [-2.  ],
        [ 0.  ]]])
>>> float(paa.distance_paa(paa_data[0], paa_data[1]))
3.15039...
>>> float(paa.distance(data[0], data[1]))
3.15039...
>>> paa.inverse_transform(paa_data)
array([[[ 0.5 ],
        [ 0.5 ],
        [-0.45],
        [-0.45],
        [ 0.  ],
        [ 0.  ]],

       [[ 2.1 ],
        [ 2.1 ],
        [-2.  ],
        [-2.  ],
        [ 0.  ],
        [ 0.  ]]])
```

Methods

distance(ts1, ts2)	Compute distance between PAA representations as defined in [1].
distance_paa(paa1, paa2)	Compute distance between PAA representations as defined in [1].
fit(X[, y])	Fit a PAA representation.
fit_transform(X[, y])	Fit a PAA representation and transform the data accordingly.
from_hdf5(path)	Load model from a HDF5 file.

continues on next page

Table 27 – continued from previous page

from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Compute time series corresponding to given PAA representations.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X[, y])	Transform a dataset of time series into its PAA representation.

distance(*ts1*, *ts2*)

Compute distance between PAA representations as defined in [1].

Parameters

ts1
[array-like] A time series

ts2
[array-like] Another time series

Returns

float
PAA distance

References

[1]

distance_paa(*paa1*, *paa2*)

Compute distance between PAA representations as defined in [1].

Parameters

paa1
[array-like] PAA representation of a time series

paa2
[array-like] PAA representation of another time series

Returns

float
PAA distance

References

[1]

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

Fit a PAA representation.

Parameters

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

Returns

PiecewiseAggregateApproximation
self

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

Fit a PAA representation and transform the data accordingly.

Parameters

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

Returns

numpy.ndarray of shape (n_ts, n_segments, d)
PAA-Transformed dataset

`classmethod from_hdf5`(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_json`(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_pickle`(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Compute time series corresponding to given PAA representations.

Parameters

X
[array-like of shape (n_ts, sz_paa, d)] A dataset of PAA series.

Returns

numpy.ndarray of shape (n_ts, sz_original_ts, d)
A dataset of time series corresponding to the provided representation.

`set_output`(*, *transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`to_hdf5`(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

`to_json`(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

`to_pickle`(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

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

Transform a dataset of time series into its PAA representation.

Parameters

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

Returns

numpy.ndarray of shape (n_ts, n_segments, d)
PAA-Transformed dataset

Examples using `tslearn.piecewise.PiecewiseAggregateApproximation`

- *PAA and SAX features*

3.10.3 SymbolicAggregateApproximation

`class tslearn.piecewise.SymbolicAggregateApproximation`(*n_segments=1*, *alphabet_size_avg=5*, *scale=False*)

Symbolic Aggregate approXimation (SAX) transformation.

SAX was originally presented in [1].

Parameters

n_segments
[int (default: 1)] Number of PAA segments to compute

alphabet_size_avg
[int (default: 5)] Number of SAX symbols to use

scale: bool (default: False)
Whether input data should be scaled for each feature to have zero mean and unit variance across the dataset passed at fit time. Default for this parameter is set to *False* in version 0.4 to ensure backward compatibility, but is likely to change in a future version.

Attributes

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

Notes

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

References

[1]

Examples

```
>>> sax = SymbolicAggregateApproximation(n_segments=3, alphabet_size_avg=2)
>>> data = [[-1., 2., 0.1, -1., 1., -1.], [1., 3.2, -1., -3., 1., -1.]]
>>> sax_data = sax.fit_transform(data)
>>> sax_data.shape
(2, 3, 1)
>>> sax_data
array([[[1],
        [0],
        [1]],

       [[1],
        [0],
        [1]]])
>>> sax.distance_sax(sax_data[0], sax_data[1])
0.0
>>> sax.distance(data[0], data[1])
0.0
>>> sax.inverse_transform(sax_data)
array([[[ 0.67448975],
        [ 0.67448975],
        [-0.67448975],
        [-0.67448975],
        [ 0.67448975],
        [ 0.67448975]],

       [[ 0.67448975],
        [ 0.67448975],
        [-0.67448975],
        [-0.67448975],
        [ 0.67448975],
        [ 0.67448975]]])
```

Methods

distance(ts1, ts2)	Compute distance between SAX representations as defined in [1].
distance_paa(paa1, paa2)	Compute distance between PAA representations as defined in [1].
distance_sax(sax1, sax2)	Compute distance between SAX representations as defined in [1].
fit(X[, y])	Fit a SAX representation.
fit_transform(X[, y])	Fit a SAX representation and transform the data accordingly.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Compute time series corresponding to given SAX representations.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
to_hdf5(path)	Save model to a HDF5 file.
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X[, y])	Transform a dataset of time series into its SAX representation.

distance(*ts1*, *ts2*)

Compute distance between SAX representations as defined in [1].

Parameters

ts1
[array-like] A time series

ts2
[array-like] Another time series

Returns

float
SAX distance

References

[1]

distance_paa(*paa1*, *paa2*)

Compute distance between PAA representations as defined in [1].

Parameters

paa1
[array-like] PAA representation of a time series

paa2
[array-like] PAA representation of another time series

Returns

float
PAA distance

References

[1]

`distance_sax`(*sax1*, *sax2*)
Compute distance between SAX representations as defined in [1].

Parameters

sax1
[array-like] SAX representation of a time series

sax2
[array-like] SAX representation of another time series

Returns

float
SAX distance

References

[1]

`fit`(*X*, *y=None*)
Fit a SAX representation.

Parameters

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

Returns

SymbolicAggregateApproximation
self

`fit_transform`(*X*, *y=None*, ***fit_params*)
Fit a SAX representation and transform the data accordingly.

Parameters

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

Returns

numpy.ndarray of integers with shape (n_ts, n_segments, d)
SAX-Transformed dataset

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

Parameters

path
[str] Full path to file.

Returns

Model instance

classmethod from_json(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

classmethod from_pickle(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

inverse_transform(*X*)

Compute time series corresponding to given SAX representations.

Parameters

X
[array-like of shape (n_ts, sz_sax, d)] A dataset of SAX series.

Returns

numpy.ndarray of shape (n_ts, sz_original_ts, d)
A dataset of time series corresponding to the provided representation.

set_output(*, *transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

`to_hdf5`(*path*)

Save model to a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

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

Raises

FileExistsError
If a file with the same path already exists.

`to_json`(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

`to_pickle`(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

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

Transform a dataset of time series into its SAX representation.

Parameters

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

Returns

numpy.ndarray of integers with shape (n_ts, n_segments, d)
SAX-Transformed dataset

Examples using `tslearn.piecewise.SymbolicAggregateApproximation`

- *PAA and SAX features*

3.11 tslearn.preprocessing

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

Classes

TimeSeriesScalerMeanVariance([mu, std, ...])	Scaler for time series datasets.
TimeSeriesScalerMinMax([value_range, ...])	Scaler for time series datasets.
TimeSeriesResampler([sz])	Resampler for time series.
TimeSeriesImputer([method, value, ...])	Missing value imputer for time series.

3.11.1 TimeSeriesScalerMeanVariance

class tslearn.preprocessing.**TimeSeriesScalerMeanVariance**(*mu=0.0*, *std=1.0*, *per_timeseries=True*, *per_feature=True*)

Scaler for time series datasets. When *per_timeseries* is False, scales features based on computation led on the fitted data, so that their mean (resp. standard deviation) in given dimensions is mu (resp. std). The transformation is stateless otherwise, dealing with each timeseries individually.

Parameters

mu
[float (default: 0.)] Mean of the output time series.

std
[float (default: 1.)] Standard deviation of the output time series.

per_timeseries: bool (default: True)
Whether the scaling should be performed per time series.

per_feature: bool (default: True)
Whether the scaling should be performed per feature. Meaningless for univariate timeseries.

Notes

NaNs within a time series are ignored when calculating mu and std.

Examples

```
>>> TimeSeriesScalerMeanVariance(mu=0.,
...                              std=1.).fit_transform([[0, 3, 6]])
array([[[-1.22474487],
        [ 0.        ],
        [ 1.22474487]]])
>>> TimeSeriesScalerMeanVariance(mu=0.,
...                              std=1.).fit_transform([[numpy.nan, 3, 6]])
array([[[nan],
        [-1.],
        [ 1.]]])
>>> TimeSeriesScalerMeanVariance(per_timeseries=False,
...                              per_feature=False
... ).fit_transform([[[1, 2], [2, 3]], [[3, 4], [4, 5]]])
array([[[-1.63299316, -0.81649658],
        [-0.81649658,  0.        ]],

       [[ 0.        ,  0.81649658],
        [ 0.81649658,  1.63299316]]])
```

Methods

fit(X[, y])	Computes the mean and standard deviation to be used for later scaling if *per_timeseries* is False.
fit_transform(X[, y])	Fit to data, then transform it.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Normalizes (mean-std) the dataset.

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

Computes the mean and standard deviation to be used for later scaling if *per_timeseries* is False. Just performs dimension checks otherwise.

Parameters

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

Returns

self

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

Fit to data, then transform it.

Parameters

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

Returns

numpy.ndarray
Resampled time series dataset.

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`set_output(*, transform=None)`

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params(**params)`

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

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

Normalizes (mean-std) the dataset. If *per_timeseries* is True, this transformation is completely stateless, and is applied to each of the timeseries individually. Otherwise, normalization is performed based on the fitted data.

Parameters

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

Returns

numpy.ndarray
Rescaled time series dataset.

Examples using `tslearn.preprocessing.TimeSeriesScalerMeanVariance`

- *DTW computation with a custom distance metric*
- *LB_Keogh*
- *Longest Common Subsequence*
- *Longest Common Subsequence with a custom distance metric*
- *sDTW multi path matching*
- *1-NN with SAX + MINDIST*
- *Kernel k-means*
- *k-means*
- *KShape*
- *Early Classification*
- *Distance and Matrix Profiles*
- *PAA and SAX features*
- *Model Persistence*

3.11.2 TimeSeriesScalerMinMax

class tslearn.preprocessing.**TimeSeriesScalerMinMax**(*value_range=(0.0, 1.0)*, *per_timeseries=True*, *per_feature=True*)

Scaler for time series datasets. When *per_timeseries* is False, scales features based on computation led on the fitted data, so that their span in given dimensions is between `min` and `max` where `value_range=(min, max)`. The transformation is stateless otherwise, dealing with each timeseries individually.

Parameters

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

per_timeseries: bool (default: True)
Wether the scaling should be performed per time series.

per_feature: bool (default: True)
Wether the scaling should be performed per feature. Meaningless for univariate timeseries.

Notes

NaNs within a time series are ignored when calculating min and max.

Examples

```
>>> TimeSeriesScalerMinMax(value_range=(1., 2.)).fit_transform([[0, 3, 6]])
array([[[1. ],
        [1.5],
        [2. ]]])
>>> TimeSeriesScalerMinMax(value_range=(1., 2.)).fit_transform(
...     [[numpy.nan, 3, 6]]
... )
array([[[nan],
        [ 1.],
        [ 2.]]])
>>> TimeSeriesScalerMinMax(value_range=(1., 2.), per_timeseries=False, per_
↪feature=False).fit_transform(
...     [[[1, 2], [2, 3]],
...     [[3, 4], [4, 5]]]
... )
array([[[1.  , 1.25],
        [1.25, 1.5 ]],

       [[1.5 , 1.75],
        [1.75, 2.  ]]])
```

Methods

fit(X[, y])	Computes the min and max to be used for later scaling if *per_timeseries* is False.
fit_transform(X[, y])	Fit to data, then transform it.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Normalizes (min-max) the dataset.

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

Computes the min and max to be used for later scaling if *per_timeseries* is False. Just performs dimension checks otherwise.

Parameters

X

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

Returns

self

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

Fit to data, then transform it.

Parameters

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

Returns

numpy.ndarray
Resampled time series dataset.

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`set_output(*, transform=None)`

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params(**params)`

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

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

Normalizes (min-max) the dataset. If *per_timeseries* is True, this transformation is completely stateless, and is applied to each of the timeseries individually. Otherwise, normalization is performed based on the fitted data.

Parameters

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

Returns

numpy.ndarray
Rescaled time series dataset.

Examples using `tslearn.preprocessing.TimeSeriesScalerMinMax`

- *Hyper-parameter tuning of a pipeline with KNeighbors time series classifier*
- *Nearest neighbors*
- *DBSCAN*
- *Soft-DTW weighted barycenters*
- *Learning Shapelets: decision boundaries in 2D distance space*
- *Aligning discovered shapelets with timeseries*
- *Learning Shapelets*
- *SVM and GAK*

3.11.3 TimeSeriesResampler

class tslearn.preprocessing.**TimeSeriesResampler**(*sz: int = -1*)

Resampler for time series. Resample time series so that they reach the target size.

Parameters

sz
[int (default: -1)] Size of the output time series. If not strictly positive, the size of the longuest timeseries in the dataset is used.

Examples

```
>>> TimeSeriesResampler(sz=5).fit_transform([[0, 3, 6]])
array([[[0. ],
        [1.5],
        [3. ],
        [4.5],
        [6. ]]])
```

Methods

fit(X[, y])	A dummy method such that it complies to the sklearn requirements.
fit_transform(X[, y])	Fit to data, then transform it.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Fit to data, then transform it.

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

A dummy method such that it complies to the sklearn requirements. Since this method is completely stateless, it just returns itself.

Parameters

X
Ignored

Returns

self

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

Fit to data, then transform it.

Parameters

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

Returns

numpy.ndarray
Resampled time series dataset.

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`set_output`(*, *transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

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

Fit to data, then transform it.

Parameters

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

Returns

numpy.ndarray
Resampled time series dataset.

Examples using `tslearn.preprocessing.TimeSeriesResampler`

- *k-means*

3.11.4 TimeSeriesImputer

class tslearn.preprocessing.**TimeSeriesImputer**(*method: str | Callable = 'mean', value: float | None = nan, keep_trailing_nans: bool = True*)

Missing value imputer for time series.

Missing values (nans) are replaced according to the chosen imputation method. There might be cases where the computation of missing values is impossible, in which case they are left unchanged (ex: mean of all nans, ffill for the first value...).

The imputer can be configured so that trailing 'empty' samples (nans for all features) are unprocessed by setting the *keep_trailing_nans* parameter to *True*. This might be handy when dealing with variable length time series datasets formatted with *to_time_series_dataset*, where time series are padded with 'empty' samples to match the length of the longest time serie. This option aims at preserving the variable length nature of the input dataset.

Time series are processed sequentially by the *transform()* and *fit_transform()* methods, and gathered using *to_time_series_dataset*, effectively padding if needed.

Parameters

method
[{'mean', 'median', 'ffill', 'bfill', 'linear', 'constant', Callable}(default: 'mean')] The method used to compute missing values.

When using linear imputation, starting nans will be replaced with first non-null value and ending nans will be replaced with last non-null value (except for 'empty' samples when *keep_trailing_nans* set to *True*).

When using a Callable, the function should take an array-like representing a timeseries with missing values as input parameter and should return the transformed timeseries.

value: float (default: nan)
The value to replace missing values with. Only used when method is *constant*.

keep_trailing_nans: bool (default: True)
Whether trailing samples with nans on all dimensions should be considered padding for variable length time series and kept unprocessed. When set to *False* , trailing 'empty' samples will be imputed.

Notes

This method allows datasets of variable lenght time series. While most missing values should be replaced, there might still be nan values in the resulting dataset representing padding when used with variable length time series, or uncomputable data.

Examples

```
>>> TimeSeriesImputer().fit_transform([[0, numpy.nan, 6]])
array([[[0.],
        [3.],
        [6.]]])
>>> # Dealing with variable length dataset
>>> TimeSeriesImputer().fit_transform([[numpy.nan, 3, 6], [numpy.nan, 3]])
array([[[4.5],
        [3. ],
        [6. ]],

       [[3. ],
```

(continues on next page)

(continued from previous page)

```
        [3. ],
        [nan]]])
>>> # Process trailing empty samples
>>> TimeSeriesImputer('ffill', keep_trailing_nans=False).fit_transform(
... [[[1, 2], [2, numpy.nan]], [[3, 4], [numpy.nan, numpy.nan]]]
... )
array([[[1., 2.],
        [2., 2.]],

       [[3., 4.],
        [3., 4.]]])
>>> # Uncomputable values are left unchanged
>>> TimeSeriesImputer('ffill').fit_transform([[numpy.nan, 3, 6]])
array([[[nan],
        [ 3.],
        [ 6.]]])
```

Methods

fit(X[, y])	A dummy method such that it complies to the sklearn requirements.
fit_transform(X[, y])	Fit to data, then transform it.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Fit to data, then transform it.

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

A dummy method such that it complies to the sklearn requirements. Since this method is completely stateless, it just returns itself.

Parameters

X
Ignored

Returns

self

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

Fit to data, then transform it.

Parameters

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

Returns

numpy.ndarray
Imputed time series dataset.

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

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

Returns

params
[dict] Parameter names mapped to their values.

`set_output(*, transform=None)`

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params(**params)`

Set the parameters of this estimator.

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

Parameters

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

Returns

self
[estimator instance] Estimator instance.

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

Fit to data, then transform it.

Parameters

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

Returns

numpy.ndarray
Imputed time series dataset

3.12 tslearn.shapelets

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

It depends on the *keras* library (Keras3+ is required) and requires *pytorch* as its computational backend. Be aware that keras backend must be configured before importing Keras, and the backend cannot be changed after the package has been imported. *tslearn* internally tries to set Keras backend, but cannot account for prior imports made in a given execution context.

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

Functions

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

3.12.1 grabocka_params_to_shapelet_size_dict

tslearn.shapelets.**grabocka_params_to_shapelet_size_dict**(*n_ts*, *ts_sz*, *n_classes*, *l*, *r*)

Compute number and length of shapelets.

This function uses the heuristic from [1].

Parameters

n_ts: int
Number of time series in the dataset

ts_sz: int
Length of time series in the dataset

n_classes: int
Number of classes in the dataset

l: float
Fraction of the length of time series to be used for base shapelet length

r: int
Number of different shapelet lengths to use

Returns

dict
Dictionary giving, for each shapelet length, the number of such shapelets to be generated

References

[1]

Examples

```
>>> d = grabocka_params_to_shapelet_size_dict(
...         n_ts=100, ts_sz=100, n_classes=3, l=0.1, r=2)
>>> keys = sorted(d.keys())
>>> print(keys)
[10, 20]
>>> print([d[k] for k in keys])
[4, 4]
```

Examples using `tslearn.shapelets.grabocka_params_to_shapelet_size_dict`

- *Learning Shapelets*

Classes

LearningShapelets([n_shapelets_per_size, ...])	Learning Time-Series Shapelets model.

3.12.2 LearningShapelets

class tslearn.shapelets.**LearningShapelets**(*n_shapelets_per_size=None*, *max_iter=10000*, *batch_size=256*, *verbose=0*, *optimizer='sgd'*, *weight_regularizer=0.0*, *shapelet_length=0.15*, *total_lengths=3*, *max_size=None*, *scale=False*, *random_state=None*)

Learning Time-Series Shapelets model.

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

From an input (possibly multidimensional) time series x and a set of shapelets $\{s_i\}_i$, the i-th coordinate of the Shapelet transform is computed as:

$$ST(x, s_i) = \min_t \sum_{\delta_t} \|x(t + \delta_t) - s_i(\delta_t)\|_2^2$$

The Shapelet model consists in a logistic regression layer on top of this transform. Shapelet coefficients as well as logistic regression weights are optimized by gradient descent on a L2-penalized cross-entropy loss.

Parameters

n_shapelets_per_size: dict (default: None)
Dictionary giving, for each shapelet size (key), the number of such shapelets to be trained (value). If None, *grabocka_params_to_shapelet_size_dict()* is used and the size used to compute is that of the shortest time series passed at fit time.

max_iter: int (default: 10,000)
Number of training epochs.

Changed in version 0.3: default value for max_iter is set to 10,000 instead of 100

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

verbose: {0, 1, 2} (default: 0)
keras verbose level.

optimizer: str or keras.optimizers.Optimizer (default: "sgd")
keras optimizer to use for training.

weight_regularizer: float (default: 0.)
Strength of the L2 regularizer to use for training the classification (softmax) layer. If 0, no regularization is performed.

shapelet_length: float (default: 0.15)
The length of the shapelets, expressed as a fraction of the time series length. Used only if *n_shapelets_per_size* is None.

total_lengths: int (default: 3)
The number of different shapelet lengths. Will extract shapelets of length i * shapelet_length for i in [1, total_lengths] Used only if *n_shapelets_per_size* is None.

max_size: int or None (default: None)
Maximum size for time series to be fed to the model. If None, it is set to the size (number of timestamps) of the training time series.

scale: bool (default: False)
Whether input data should be scaled for each feature of each time series to lie in the [0-1] interval. Default for this parameter is set to *False* in version 0.4 to ensure backward compatibility, but is likely to change in a future version.

random_state
[int or None, optional (default: None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Attributes

shapelets_
[numpy.ndarray of objects, each object being a time series] Set of time-series shapelets.

shapelets_as_time_series_
[numpy.ndarray of shape (n_shapelets, sz_shp, d)] Set of time-series shapelets formatted as a `tslearn` time series dataset.

where `sz_shp` is the maximum of all shapelet sizes
Set of time-series shapelets formatted as a `tslearn` time series dataset.

transformer_model_
[keras.Model] Transforms an input dataset of timeseries into distances to the learned shapelets.

locator_model_
[keras.Model] Returns the indices where each of the shapelets can be found (minimal distance) within each of the timeseries of the input dataset.

model_
[keras.Model] Directly predicts the class probabilities for the input timeseries.

history_
[dict] Dictionary of losses and metrics recorded during fit.

Notes

This model does not support HDF5 serialization.

References

[1]

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=16, d=2, n_blobs=3)
>>> clf = LearningShapelets(n_shapelets_per_size={4: 5},
...                         max_iter=1, verbose=0)
>>> clf.fit(X, y).shapelets_.shape
(5,)
>>> clf.shapelets_[0].shape
(4, 2)
>>> clf.predict(X).shape
(30,)
>>> clf.predict_proba(X).shape
(30, 3)
>>> clf.transform(X).shape
(30, 5)
```

Methods

fit(X, y)	Learn time-series shapelets.
fit_transform(X[, y])	Fit to data, then transform it.
from_hdf5(path)	Load model from a HDF5 file.
from_json(path)	Load model from a JSON file.
from_pickle(path)	Load model from a pickle file.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
get_weights([layer_name])	Return model weights (or weights for a given layer if *layer_name* is provided).
locate(X)	Compute shapelet match location for a set of time series.
predict(X)	Predict class for a given set of time series.
predict_proba(X)	Predict class probability for a given set of time series.
score(X, y[, sample_weight])	Return accuracy on provided data and labels.
set_output(*[, transform])	Set output container.
set_params(**params)	Set the parameters of this estimator.
set_score_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `score` method.
set_weights(weights[, layer_name])	Set model weights (or weights for a given layer if *layer_name* is provided).
to_hdf5(path)	LearningShapelet is not HDF5 serializable
to_json(path)	Save model to a JSON file.
to_pickle(path)	Save model to a pickle file.
transform(X)	Generate shapelet transform for a set of time series.

`fit`(*X*, *y*)

Learn time-series shapelets.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

y
[array-like of shape=(n_ts,)] Time series labels.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters. Pass only if the estimator accepts additional params in its *fit* method.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`classmethod from_hdf5`(*path*)

Load model from a HDF5 file. Requires `h5py` http://docs.h5py.org/

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_json`(*path*)

Load model from a JSON file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`classmethod from_pickle`(*path*)

Load model from a pickle file.

Parameters

path
[str] Full path to file.

Returns

Model instance

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_weights`(*layer_name=None*)

Return model weights (or weights for a given layer if *layer_name* is provided).

Parameters

layer_name: str or None (default: None)
Name of the layer for which weights should be returned. If None, all model weights are returned. Available layer names with weights are:

- "shapelets_i_j" with i an integer for the shapelet id and j an integer for the dimension
- "classification" for the final classification layer

Returns

list
list of model (or layer) weights

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=100, sz=256, d=1, n_blobs=3)
>>> clf = LearningShapelets(n_shapelets_per_size={10: 5}, max_iter=1,
...                         verbose=0)
>>> clf.fit(X, y).get_weights("classification")[0].shape
(5, 3)
>>> clf.get_weights("shapelets_0")[0].shape
(5, 10, 1)
>>> len(clf.get_weights("shapelets_0"))
1
```

`locate`(*X*)

Compute shapelet match location for a set of time series.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_shapelets)
Location of the shapelet matches for the provided time series.

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X = numpy.zeros((3, 10, 1))
>>> X[0, 4:7, 0] = numpy.array([1, 2, 3])
>>> y = [1, 0, 0]
>>> # Data is all zeros except a motif 1-2-3 in the first time series
>>> clf = LearningShapelets(n_shapelets_per_size={3: 1}, max_iter=1,
...                         verbose=0)
>>> _ = clf.fit(X, y)
>>> weights_shapelet = [
...     numpy.array([[[1], [2], [3]]])
... ]
>>> clf.set_weights(weights_shapelet, layer_name="shapelets_0")
>>> clf.locate(X)
array([[4],
       [0],
       [0]])
```

predict(*X*)

Predict class for a given set of time series.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts,) or (n_ts, n_classes), depending on the shape of the label vector provided at training time.
Index of the cluster each sample belongs to or class probability matrix, depending on what was provided at training time.

predict_proba(*X*)

Predict class probability for a given set of time series.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_classes),
Class probability matrix.

score(*X*, *y*, *sample_weight=None*)

Return accuracy on provided data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` w.r.t. *y*.

`set_output`(*, *transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas", "polars"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *"polars"*: Polars output
- *None*: Transform configuration is unchanged

Added in version 1.4: *"polars"* option was added.

Returns

self
[estimator instance] Estimator instance.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *LearningShapelets*

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

Returns

self
[object] The updated object.

set_weights(*weights*, *layer_name=None*)

Set model weights (or weights for a given layer if *layer_name* is provided).

Parameters

weights: list of ndarrays
Weights to set for the model / target layer

layer_name: str or None (default: None)
Name of the layer for which weights should be set. If None, all model weights are set. Available layer names with weights are:

- "shapelets_i_j" with i an integer for the shapelet id and j an integer for the dimension
- "classification" for the final classification layer

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=16, d=1, n_blobs=3)
>>> clf = LearningShapelets(n_shapelets_per_size={3: 1}, max_iter=1,
...                         verbose=0)
>>> _ = clf.fit(X, y)
>>> weights_shapelet = [
...     numpy.array([[[1], [2], [3]]])
... ]
>>> clf.set_weights(weights_shapelet, layer_name="shapelets_0")
```

(continues on next page)

(continued from previous page)

```
>>> clf.shapelets_as_time_series_
array([[[1.],
        [2.],
        [3.]]])
```

property shapelets_as_time_series_

Set of time-series shapelets formatted as a `tslearn` time series dataset.

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=256, d=1, n_blobs=3)
>>> model = LearningShapelets(n_shapelets_per_size={3: 2, 4: 1},
...                           max_iter=1)
>>> _ = model.fit(X, y)
>>> model.shapelets_as_time_series_.shape
(3, 4, 1)
```

to_hdf5(*path*)

LearningShapelet is not HDF5 serializable

to_json(*path*)

Save model to a JSON file.

Parameters

path
[str] Full file path.

to_pickle(*path*)

Save model to a pickle file.

Parameters

path
[str] Full file path.

transform(*X*)

Generate shapelet transform for a set of time series.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_shapelets)
Shapelet-Transform of the provided time series.

Examples using `tslearn.shapelets.LearningShapelets`

- *Learning Shapelets: decision boundaries in 2D distance space*
- *Aligning discovered shapelets with timeseries*
- *Learning Shapelets*

3.13 tslearn.svm

The *tslearn.svm* module contains Support Vector Classifier (SVC) and Support Vector Regressor (SVR) models for time series.

Classes

TimeSeriesSVC([C, kernel, degree, gamma, ...])	Time-series specific Support Vector Classifier.
TimeSeriesSVR([C, kernel, degree, gamma, ...])	Time-series specific Support Vector Regressor.

3.13.1 TimeSeriesSVC

class tslearn.svm.**TimeSeriesSVC**(*C=1.0*, *kernel='gak'*, *degree=3*, *gamma='auto'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *n_jobs=None*, *verbose=0*, *max_iter=-1*, *decision_function_shape='ovr'*, *random_state=None*)

Time-series specific Support Vector Classifier.

Parameters

C
[float, optional (default=1.0)] Penalty parameter C of the error term.

kernel
[string, optional (default='gak')] Specifies the kernel type to be used in the algorithm. It must be one of 'gak' or a kernel accepted by `sklearn.svm.SVC`. If none is given, 'gak' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

degree
[int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma
[float, optional (default='auto')] Kernel coefficient for 'gak', 'rbf', 'poly' and 'sigmoid'. For 'gak' kernel, a *RuntimeError* is raised at fit time when value is close to 0 and therefore not compatible with 'gak' kernel.

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*). A *RuntimeError* is raised at fit time when computed value is close to 0 and therefore not compatible with 'gak' kernel.
- for other kernels (eg. 'rbf'), 1/n_features will be used.

coef0
[float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking
[boolean, optional (default=True)] Whether to use the shrinking heuristic.

probability
[boolean, optional (default=False)] Whether to enable probability estimates. This must be enabled prior to calling *fit*, and will slow down that method. Also, probability estimates are not guaranteed to match predict output. See our *dedicated user guide section* for more details.

tol
[float, optional (default=1e-3)] Tolerance for stopping criterion.

cache_size
[float, optional (default=200.0)] Specify the size of the kernel cache (in MB).

class_weight
[{dict, 'balanced'}, optional] Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

n_jobs
[int or None, optional (default=None)] The number of jobs to run in parallel for GAK cross-similarity matrix computations. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose
[int, default: 0] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter
[int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape
['ovo', 'ovr', default='ovr'] Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2).

random_state
[int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Attributes

support_
[array-like, shape = [n_SV]] Indices of support vectors.

n_support_
[array-like, dtype=int32, shape = [n_class]] Number of support vectors for each class.

support_vectors_
[list of arrays of shape [n_SV, sz, d]] List of support vectors in tslearn dataset format, one array per class

dual_coef_
[array, shape = [n_class-1, n_SV]] Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide of `sklearn` for details.

coef_
[array, shape = [n_class-1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel. *coef_* is a readonly property derived from *dual_coef_* and *support_vectors_*.

intercept_
[array, shape = [n_class * (n_class-1) / 2]] Constants in decision function.

svm_estimator_
[sklearn.svm.SVC] The underlying sklearn estimator

References

Fast Global Alignment Kernels. Marco Cuturi. ICML 2011.

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=64, d=2, n_blobs=2)
>>> clf = TimeSeriesSVC(kernel="gak", gamma="auto", probability=True)
>>> clf.fit(X, y).predict(X).shape
(20,)
>>> sv = clf.support_vectors_
>>> len(sv)  # should be equal to the nr of classes in the clf problem
2
>>> sv[0].shape
(..., 64, 2)
>>> sv_sum = sum([sv_i.shape[0] for sv_i in sv])
>>> bool(sv_sum == clf.svm_estimator_.n_support_.sum())
True
>>> clf.decision_function(X).shape
(20,)
>>> clf.predict_log_proba(X).shape
(20, 2)
>>> clf.predict_proba(X).shape
(20, 2)
```

Methods

decision_function(X)	Evaluates the decision function for the samples in X.
fit(X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class for a given set of time series.
predict_log_proba(X)	Predict class log-probabilities for a given set of time series.
predict_proba(X)	Predict class probability for a given set of time series.
score(X, y[, sample_weight])	Return accuracy on provided data and labels.
set_fit_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `fit` method.
set_params(**params)	Set the parameters of this estimator.
set_score_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `score` method.

decision_function(*X*)

Evaluates the decision function for the samples in X.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

ndarray of shape (n_samples, n_classes * (n_classes-1) / 2)
Returns the decision function of the sample for each class in the model. If decision_function_shape='ovr', the shape is (n_samples, n_classes).

`fit`(*X*, *y*, *sample_weight=None*)

Fit the SVM model according to the given training data.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

y
[array-like of shape=(n_ts,)] Time series labels.

sample_weight
[array-like of shape (n_samples,), default=None] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Predict class for a given set of time series.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts,) or (n_ts, n_classes), depending on the shape of the label vector provided at training time.
Index of the cluster each sample belongs to or class probability matrix, depending on what was provided at training time.

`predict_log_proba`(*X*)

Predict class log-probabilities for a given set of time series.

Note that probability estimates are not guaranteed to match predict output. See our *dedicated user guide section* for more details.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_classes),
Class probability matrix.

`predict_proba`(*X*)

Predict class probability for a given set of time series.

Note that probability estimates are not guaranteed to match predict output. See our *dedicated user guide section* for more details.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts, n_classes),
Class probability matrix.

`score`(*X*, *y*, *sample_weight=None*)

Return accuracy on provided data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` w.r.t. *y*.

`set_fit_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesSVC*

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.

- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `fit`.

Returns

self
[object] The updated object.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesSVC*

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

Returns

self
[object] The updated object.

Examples using `tslearn.svm.TimeSeriesSVC`

- *SVM and GAK*

3.13.2 TimeSeriesSVR

class tslearn.svm.**TimeSeriesSVR**(*C=1.0*, *kernel='gak'*, *degree=3*, *gamma='auto'*, *coef0=0.0*, *tol=0.001*, *epsilon=0.1*, *shrinking=True*, *cache_size=200*, *n_jobs=None*, *verbose=0*, *max_iter=-1*, *random_state=None*)

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'. For 'gak' kernel, a *RuntimeError* is raised at fit time when value is close to 0 and therefore not compatible with 'gak' kernel.

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*). A *RuntimeError* is raised at fit time when computed value is close to 0 and therefore not compatible with 'gak' kernel.
- for other kernels (eg. 'rbf'), 1/n_features will be used.

coef0
[float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

tol
[float, optional (default=1e-3)] Tolerance for stopping criterion.

epsilon
[float, optional (default=0.1)] Epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.

shrinking
[boolean, optional (default=True)] Whether to use the shrinking heuristic.

cache_size
[float, optional (default=200.0)] Specify the size of the kernel cache (in MB).

n_jobs
[int or None, optional (default=None)] The number of jobs to run in parallel for GAK cross-similarity matrix computations. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learns' Glossary for more details.

verbose
[int, default: 0] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter
[int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

random_state
[int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Attributes

support_
[array-like, shape = [n_SV]] Indices of support vectors.

support_vectors_
[array of shape [n_SV, sz, d]] Support vectors in tslearn dataset format

dual_coef_
[array, shape = [1, n_SV]] Coefficients of the support vector in the decision function.

coef_
[array, shape = [1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel. *coef_* is readonly property derived from *dual_coef_* and *support_vectors_*.

intercept_
[array, shape = [1]] Constants in decision function.

svm_estimator_
[sklearn.svm.SVR] The underlying sklearn estimator

References

Fast Global Alignment Kernels. Marco Cuturi. ICML 2011.

Examples

```
>>> from tslearn.generators import random_walk_blobs
>>> X, y = random_walk_blobs(n_ts_per_blob=10, sz=64, d=2, n_blobs=2)
>>> import numpy
>>> y = y.astype(float) + numpy.random.randn(20) * .1
>>> reg = TimeSeriesSVR(kernel="gak", gamma="auto")
>>> reg.fit(X, y).predict(X).shape
```

(continues on next page)

(continued from previous page)

```
(20,)
>>> sv = reg.support_vectors_
>>> sv.shape
(..., 64, 2)
>>> sv.shape[0] <= 20
True
```

Methods

fit(X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_metadata_routing()	Get metadata routing of this object.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class for a given set of time series.
score(X, y[, sample_weight])	Return coefficient of determination on test data.
set_fit_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `fit` method.
set_params(**params)	Set the parameters of this estimator.
set_score_request(*[, sample_weight])	Configure whether metadata should be requested to be passed to the `score` method.

fit(*X*, *y*, *sample_weight=None*)

Fit the SVM model according to the given training data.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

y
[array-like of shape=(n_ts,)] Time series labels.

sample_weight
[array-like of shape (n_samples,), default=None] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Predict class for a given set of time series.

Parameters

X
[array-like of shape=(n_ts, sz, d)] Time series dataset.

Returns

array of shape=(n_ts,) or (n_ts, dim_output), depending on the shape of the target vector provided at training time.
Predicted targets

`score`(*X*, *y*, *sample_weight=None*)

Return coefficient of determination on test data.

The coefficient of determination, R^2, is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares `((y_true - y_pred)** 2).sum()` and v is the total sum of squares `((y_true - y_true.mean()) ** 2).sum()`. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] R^2 of `self.predict(X)` w.r.t. *y*.

Notes

The R^2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

`set_fit_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesSVR*

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.

- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `fit`.

Returns

self
[object] The updated object.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TimeSeriesSVR*

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

Returns

self
[object] The updated object.

3.14 tslearn.utils

The *tslearn.utils* module includes various utilities.

Generic functions

to_time_series(ts[, remove_nans, be, dtype])	Transforms a time series so that it fits the format used in `tslearn` models.
to_time_series_dataset(dataset[, dtype, be])	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[, be])	Returns actual time series size.
ts_zeros(sz[, d])	Returns a time series made of zero values.
load_time_series_txt(fname)	Loads a time series dataset from disk.
save_time_series_txt(fname, dataset[, fmt])	Writes a time series dataset to disk.
check_equal_size(dataset[, be])	Check if all time series in the dataset have the same size.
check_dims(X[, X_fit_dims, extend, ...])	Reshapes X to a 3-dimensional array of X.shape[0] univariate timeseries of length X.shape[1] if X is 2-dimensional and extend is True.

3.14.1 to_time_series

`tslearn.utils.`**`to_time_series`**(*ts*, *remove_nans=False*, *be=None*, *dtype=<class 'float'>*)

Transforms a time series so that it fits the format used in `tslearn` models.

Parameters

ts
[array-like, shape=(sz, d) or (sz,)] The time series to be transformed. If shape is (sz,), the time series is assumed to be univariate.

remove_nans
[bool (default: False)] Whether trailing NaNs at the end of the time series should be removed or not

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

dtype
[data type (default: float)] Data type for the returned time series, depending on the backend.

Returns

ts_out
[array-like, shape=(sz, d)] The transformed time series. This is always guaranteed to be a new time series and never just a view into the old one.

See also

to_time_series_dataset
Transforms a dataset of time series

Examples

```
>>> to_time_series([1, 2])
array([[1.],
       [2.]])
>>> to_time_series([1, 2, numpy.nan])
array([[ 1.],
       [ 2.],
       [nan]])
>>> to_time_series([1, 2, numpy.nan], remove_nans=True)
array([[1.],
       [2.]])
```

3.14.2 to_time_series_dataset

`tslearn.utils.`**`to_time_series_dataset`**(*dataset*, *dtype=<class 'float'>*, *be=None*)

Transforms a time series dataset so that it fits the format used in `tslearn` models.

Parameters

dataset
[array-like, shape=(n_ts, sz, d) or (n_ts, sz) or (sz,)] The dataset of time series to be transformed. A single time series will be automatically wrapped into a dataset with a single entry.

dtype
[data type (default: float)] Data type for the returned dataset, depending on the backend.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

dataset_out
[array-like, 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])
array([[[1.],
        [2.]]])
>>> to_time_series_dataset([[1, 2], [1, 4, 3]])
array([[[ 1.],
        [ 2.],
        [nan]],

       [[ 1.],
        [ 4.],
        [ 3.]]])
>>> to_time_series_dataset([]).shape
(0, 0, 0)
```

3.14.3 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: float64)] Data type for the returned dataset.

return_dim
[boolean (optional, default: False)] Whether the dimensionality (third dimension should be returned together with the transformed dataset).

Returns

numpy.ndarray of shape (n_ts, sz * d)
The transformed dataset of time series.

int (optional, if return_dim=True)
The dimensionality of the original tslearn dataset (third dimension)

See also

to_time_series_dataset
Transforms a time series dataset to `tslearn` **format.**

Examples

```
>>> to_sklearn_dataset([[1, 2]], return_dim=True)
(array([[1., 2.]]), 1)
>>> to_sklearn_dataset([[1, 2], [1, 4, 3]])
array([[ 1.,  2., nan],
       [ 1.,  4.,  3.]])
```

3.14.4 ts_size

`tslearn.utils.`**ts_size**(*ts*, *be=None*)

Returns actual time series size.

Final timesteps that have *NaN* values for all dimensions will be removed from the count. Infinity and negative infinity ar considered valid time series values.

Parameters

ts
[array-like] A time series.

be
[Backend object or string or None] Backend. If *be* is an instance of the class *NumPyBackend* or the string *"numpy"*, the NumPy backend is used. If *be* is an instance of the class *PyTorchBackend* or the string *"pytorch"*, the PyTorch backend is used. If *be* is *None*, the backend is determined by the input arrays. See our *dedicated user-guide page* for more information.

Returns

int
Actual size of the time series.

Examples

```
>>> ts_size([1, 2, 3, numpy.nan])
3
>>> ts_size([1, numpy.nan])
1
>>> ts_size([numpy.nan])
0
>>> ts_size([[1, 2],
...          [2, 3],
...          [3, 4],
...          [numpy.nan, 2],
...          [numpy.nan, numpy.nan]])
4
>>> ts_size([numpy.nan, 3, numpy.inf, numpy.nan])
3
```

Examples using `tslearn.utils.ts_size`

- *Learning Shapelets*

3.14.5 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)
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> ts_zeros(5).shape
(5, 1)
```

3.14.6 load_time_series_txt

`tslearn.utils.`**`load_time_series_txt`**(*fname*)

Loads a time series dataset from disk.

Parameters

fname
[string] Path to the file from which time series should be read.

Returns

numpy.ndarray or array of numpy.ndarray
The dataset of time series.

See also

`save_time_series_txt`
Save time series to disk

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4], [1, 2, 3]])
>>> save_time_series_txt("tmp-tslearn-test.txt", dataset)
>>> reloaded_dataset = load_time_series_txt("tmp-tslearn-test.txt")
```

3.14.7 save_time_series_txt

tslearn.utils.**save_time_series_txt**(*fname*, *dataset*, *fmt='%.18e'*)

Writes a time series dataset to disk.

Parameters

fname
[string] Path to the file in which time series should be written.

dataset
[array-like] The dataset of time series to be saved.

fmt
[string (default: "%.18e")] Format to be used to write each value.

See also

load_time_series_txt
Load time series from disk

Examples

```
>>> dataset = to_time_series_dataset([[1, 2, 3, 4], [1, 2, 3]])
>>> save_time_series_txt("tmp-tslearn-test.txt", dataset)
```

3.14.8 check_equal_size

tslearn.utils.**check_equal_size**(*dataset*, *be=None*)

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
>>> check_equal_size([])
True
```

3.14.9 check_dims

tslearn.utils.**check_dims**(*X*, *X_fit_dims=None*, *extend=True*, *check_n_features_only=False*)

Reshapes X to a 3-dimensional array of X.shape[0] univariate timeseries of length X.shape[1] if X is 2-dimensional and extend is True. Then checks whether the provided X_fit_dims and the dimensions of X (except for the first one), match.

Parameters

X
[array-like] The first array to be compared.

X_fit_dims
[tuple (default: None)] The dimensions of the data generated by fit, to compare with the dimensions of the provided array X. If None, then only perform reshaping of X, if necessary.

extend
[boolean (default: True)] Whether to reshape X, if it is 2-dimensional.

check_n_features_only: boolean (default: False)

Returns

array
Reshaped X array

Raises

ValueError
Will raise exception if X is None or (if X_fit_dims is provided) one of the dimensions of the provided data, except the first, does not match X_fit_dims.

Examples

```
>>> X = numpy.empty((10, 3))
>>> check_dims(X).shape
(10, 3, 1)
>>> X = numpy.empty((10, 3, 1))
>>> check_dims(X).shape
(10, 3, 1)
>>> X_fit_dims = (5, 3, 1)
>>> check_dims(X, X_fit_dims).shape
(10, 3, 1)
>>> X_fit_dims = (5, 3, 2)
>>> check_dims(X, X_fit_dims)
Traceback (most recent call last):
ValueError: Dimensions (except first) must match! ((5, 3, 2) and (10, 3, 1)
are passed shapes)
>>> X_fit_dims = (5, 5, 1)
>>> check_dims(X, X_fit_dims, check_n_features_only=True).shape
(10, 3, 1)
>>> X_fit_dims = (5, 5, 2)
>>> check_dims(
...     X,
...     X_fit_dims,
...     check_n_features_only=True
... )
Traceback (most recent call last):
ValueError: Number of features of the provided timeseries must match!
```

(continues on next page)

(continued from previous page)

```
(last dimension) must match the one of the fitted data!
((5, 5, 2) and (10, 3, 1) are passed shapes)
```

Conversion functions

The following functions are provided for the sake of interoperability between standard Python packages for time series. They allow conversion between *tslearn* format and other libraries' formats.

to_pyts_dataset(X)	Transform a tslearn-compatible dataset into a pyts dataset.
from_pyts_dataset(X)	Transform a pyts-compatible dataset into a tslearn dataset.
to_sktime_dataset(X)	Transform a tslearn-compatible dataset into a sktime dataset.
from_sktime_dataset(X)	Transform a sktime-compatible dataset into a tslearn dataset.
to_cesium_dataset(X)	Transform a tslearn-compatible dataset into a cesium dataset.
from_cesium_dataset(X)	Transform a cesium-compatible dataset into a tslearn dataset.
to_seglearn_dataset(X)	Transform a tslearn-compatible dataset into a seglearn dataset.
from_seglearn_dataset(X)	Transform a seglearn-compatible dataset into a tslearn dataset.
to_tsfresh_dataset(X)	Transform a tslearn-compatible dataset into a tsfresh dataset.
from_tsfresh_dataset(X)	Transform a tsfresh-compatible dataset into a tslearn dataset.
to_stumpy_dataset(X)	Transform a tslearn-compatible dataset into a stumpy dataset.
from_stumpy_dataset(X)	Transform a stumpy-compatible dataset into a tslearn dataset.
to_pyflux_dataset(X)	Transform a tslearn-compatible dataset into a pyflux dataset.
from_pyflux_dataset(X)	Transform a pyflux-compatible dataset into a tslearn dataset.

3.14.10 to_pyts_dataset

tslearn.utils.**to_pyts_dataset**(*X*)

Transform a tslearn-compatible dataset into a pyts dataset.

Parameters

X: array, shape = (n_ts, sz, d)
tslearn-formatted dataset to be cast to pyts format

Returns

array, shape=(n_ts, sz) if d=1, (n_ts, d, sz) otherwise
pyts-formatted dataset

Examples

```
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> pyts_arr = to_pyts_dataset(tslearn_arr)
>>> pyts_arr.shape
(10, 16)
>>> tslearn_arr = numpy.random.randn(10, 16, 2)
>>> pyts_arr = to_pyts_dataset(tslearn_arr)
>>> pyts_arr.shape
(10, 2, 16)
>>> tslearn_arr = [numpy.random.randn(16, 1), numpy.random.randn(10, 1)]
>>> to_pyts_dataset(tslearn_arr)
Traceback (most recent call last):
...
ValueError: All the time series in the array should be of equal lengths
```

3.14.11 from_pyts_dataset

tslearn.utils.**from_pyts_dataset**(*X*)

Transform a pyts-compatible dataset into a tslearn dataset.

Parameters

X: array, shape = (n_ts, sz) or (n_ts, d, sz)
pyts-formatted dataset

Returns

array, shape=(n_ts, sz, d)
tslearn-formatted dataset

Examples

```
>>> pyts_arr = numpy.random.randn(10, 16)
>>> tslearn_arr = from_pyts_dataset(pyts_arr)
>>> tslearn_arr.shape
(10, 16, 1)
>>> pyts_arr = numpy.random.randn(10, 2, 16)
>>> tslearn_arr = from_pyts_dataset(pyts_arr)
>>> tslearn_arr.shape
(10, 16, 2)
>>> pyts_arr = numpy.random.randn(10)
>>> from_pyts_dataset(pyts_arr)
Traceback (most recent call last):
...
ValueError: X is not a valid input pyts array.
```

3.14.12 to_sktime_dataset

tslearn.utils.**to_sktime_dataset**(*X*)

Transform a tslearn-compatible dataset into a sktime dataset.

Parameters

X: array, shape = (n_ts, sz, d)
tslearn-formatted dataset to be cast to sktime format

Returns

Pandas data-frame
sktime-formatted dataset

Notes

Conversion from/to sktime format requires pandas to be installed.

Examples

```
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> sktime_arr = to_sktime_dataset(tslearn_arr)
>>> sktime_arr.shape
(10, 1)
>>> sktime_arr["dim_0"][0].shape
(16,)
>>> tslearn_arr = numpy.random.randn(10, 16, 2)
>>> sktime_arr = to_sktime_dataset(tslearn_arr)
>>> sktime_arr.shape
(10, 2)
>>> sktime_arr["dim_1"][0].shape
(16,)
```

3.14.13 from_sktime_dataset

`tslearn.utils.`**`from_sktime_dataset`**`(X)`

Transform a sktime-compatible dataset into a tslearn dataset.

Parameters

X: pandas data-frame
sktime-formatted dataset

Returns

array, shape=(n_ts, sz, d)
tslearn-formatted dataset

Notes

Conversion from/to sktime format requires pandas to be installed.

Examples

```
>>> import pandas as pd
>>> sktime_df = pd.DataFrame()
>>> sktime_df["dim_0"] = [pd.Series([1, 2, 3]), pd.Series([4, 5, 6])]
>>> tslearn_arr = from_sktime_dataset(sktime_df)
>>> tslearn_arr.shape
(2, 3, 1)
>>> sktime_df = pd.DataFrame()
>>> sktime_df["dim_0"] = [pd.Series([1, 2, 3]),
...                       pd.Series([4, 5, 6, 7])]
>>> sktime_df["dim_1"] = [pd.Series([8, 9, 10]),
...                       pd.Series([11, 12, 13, 14])]
```

(continues on next page)

(continued from previous page)

```
>>> tslearn_arr = from_sktime_dataset(sktime_df)
>>> tslearn_arr.shape
(2, 4, 2)
>>> sktime_arr = numpy.random.randn(10, 1, 16)
>>> from_sktime_dataset(
...     sktime_arr
... )
Traceback (most recent call last):
...
ValueError: X is not a valid input sktime array.
```

3.14.14 to_cesium_dataset

tslearn.utils.**to_cesium_dataset**(*X*)

Transform a tslearn-compatible dataset into a cesium dataset.

Parameters

X: array, shape = (n_ts, sz, d), where n_ts=1
tslearn-formatted dataset to be cast to cesium format

Returns

list of cesium TimeSeries
cesium-formatted dataset (cf. link)

Notes

Conversion from/to cesium format requires cesium to be installed.

Examples

```
>>> tslearn_arr = numpy.random.randn(3, 16, 1)
>>> cesium_ds = to_cesium_dataset(tslearn_arr)
>>> len(cesium_ds)
3
>>> cesium_ds[0].measurement.shape
(16,)
>>> tslearn_arr = numpy.random.randn(3, 16, 2)
>>> cesium_ds = to_cesium_dataset(tslearn_arr)
>>> len(cesium_ds)
3
>>> cesium_ds[0].measurement.shape
(2, 16)
>>> tslearn_arr = [[1, 2, 3], [1, 2, 3, 4]]
>>> cesium_ds = to_cesium_dataset(tslearn_arr)
>>> len(cesium_ds)
2
>>> cesium_ds[0].measurement.shape
(3,)
```

3.14.15 from_cesium_dataset

tslearn.utils.**from_cesium_dataset**(*X*)

Transform a cesium-compatible dataset into a tslearn dataset.

Parameters

X: list of cesium TimeSeries
cesium-formatted dataset (cf. link)

Returns

array, shape=(n_ts, sz, d)
tslearn-formatted dataset.

Notes

Conversion from/to cesium format requires cesium to be installed.

Examples

```
>>> from cesium.time_series import TimeSeries
>>> cesium_ds = [TimeSeries(m=numpy.array([1, 2, 3, 4]))]
>>> tslearn_arr = from_cesium_dataset(cesium_ds)
>>> tslearn_arr.shape
(1, 4, 1)
>>> cesium_ds = [
...     TimeSeries(m=numpy.array([[1, 2, 3, 4],
...                               [5, 6, 7, 8]]))
... ]
>>> tslearn_arr = from_cesium_dataset(cesium_ds)
>>> tslearn_arr.shape
(1, 4, 2)
```

3.14.16 to_seglearn_dataset

tslearn.utils.**to_seglearn_dataset**(*X*)

Transform a tslearn-compatible dataset into a seglearn dataset.

Parameters

X: array, shape = (n_ts, sz, d)
tslearn-formatted dataset to be cast to seglearn format

Returns

array of arrays, shape=(n_ts,)
seglearn-formatted dataset. i-th sub-array in the list has shape (sz_i, d)

Examples

```
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> seglearn_arr = to_seglearn_dataset(tslearn_arr)
>>> seglearn_arr.shape
(10, 16, 1)
>>> tslearn_arr = numpy.random.randn(10, 16, 2)
>>> seglearn_arr = to_seglearn_dataset(tslearn_arr)
```

(continues on next page)

(continued from previous page)

```
>>> seglearn_arr.shape
(10, 16, 2)
>>> tslearn_arr = [numpy.random.randn(16, 2), numpy.random.randn(10, 2)]
>>> seglearn_arr = to_seglearn_dataset(tslearn_arr)
>>> seglearn_arr.shape
(2,)
>>> seglearn_arr[0].shape
(16, 2)
>>> seglearn_arr[1].shape
(10, 2)
```

3.14.17 from_seglearn_dataset

tslearn.utils.**from_seglearn_dataset**(*X*)

Transform a seglearn-compatible dataset into a tslearn dataset.

Parameters

X: list of arrays, or array of arrays, shape = (n_ts,)
seglearn-formatted dataset. i-th sub-array in the list has shape (sz_i, d)

Returns

array, shape=(n_ts, sz, d), where sz is the maximum of all array lengths
tslearn-formatted dataset

Examples

```
>>> seglearn_arr = [numpy.random.randn(10, 1), numpy.random.randn(10, 1)]
>>> tslearn_arr = from_seglearn_dataset(seglearn_arr)
>>> tslearn_arr.shape
(2, 10, 1)
>>> seglearn_arr = [numpy.random.randn(10, 1), numpy.random.randn(5, 1)]
>>> tslearn_arr = from_seglearn_dataset(seglearn_arr)
>>> tslearn_arr.shape
(2, 10, 1)
>>> seglearn_arr = numpy.random.randn(2, 10, 1)
>>> tslearn_arr = from_seglearn_dataset(seglearn_arr)
>>> tslearn_arr.shape
(2, 10, 1)
```

3.14.18 to_tsfresh_dataset

tslearn.utils.**to_tsfresh_dataset**(*X*)

Transform a tslearn-compatible dataset into a tsfresh dataset.

Parameters

X: array, shape = (n_ts, sz, d)
tslearn-formatted dataset to be cast to tsfresh format

Returns

Pandas data-frame
tsfresh-formatted dataset ("flat" data frame, as described there)

Notes

Conversion from/to tsfresh format requires pandas to be installed.

Examples

```
>>> tslearn_arr = numpy.random.randn(1, 16, 1)
>>> tsfresh_df = to_tsfresh_dataset(tslearn_arr)
>>> tsfresh_df.shape
(16, 3)
>>> tslearn_arr = numpy.random.randn(1, 16, 2)
>>> tsfresh_df = to_tsfresh_dataset(tslearn_arr)
>>> tsfresh_df.shape
(16, 4)
```

3.14.19 from_tsfresh_dataset

tslearn.utils.**from_tsfresh_dataset**(*X*)

Transform a tsfresh-compatible dataset into a tslearn dataset.

Parameters

X: pandas data-frame
tsfresh-formatted dataset ("flat" data frame, as described there)

Returns

array, shape=(n_ts, sz, d)
tslearn-formatted dataset. Column order is kept the same as in the original data frame.

Notes

Conversion from/to tsfresh format requires pandas to be installed.

Examples

```
>>> import pandas as pd
>>> tsfresh_df = pd.DataFrame(columns=["id", "time", "a", "b"])
>>> tsfresh_df["id"] = [0, 0, 0]
>>> tsfresh_df["time"] = [0, 1, 2]
>>> tsfresh_df["a"] = [-1, 4, 7]
>>> tsfresh_df["b"] = [8, -3, 2]
>>> tslearn_arr = from_tsfresh_dataset(tsfresh_df)
>>> tslearn_arr.shape
(1, 3, 2)
>>> tsfresh_df = pd.DataFrame(columns=["id", "time", "a"])
>>> tsfresh_df["id"] = [0, 0, 0, 1, 1]
>>> tsfresh_df["time"] = [0, 1, 2, 0, 1]
>>> tsfresh_df["a"] = [-1, 4, 7, 9, 1]
>>> tslearn_arr = from_tsfresh_dataset(tsfresh_df)
>>> tslearn_arr.shape
(2, 3, 1)
>>> tsfresh_df = numpy.random.randn(10, 1, 16)
>>> from_tsfresh_dataset(
...     tsfresh_df
```

(continues on next page)

(continued from previous page)

```
... )
Traceback (most recent call last):
...
ValueError: X is not a valid input tsfresh array.
```

3.14.20 to_stumpy_dataset

tslearn.utils.**to_stumpy_dataset**(*X*)

Transform a tslearn-compatible dataset into a stumpy dataset.

Parameters

X: array, shape = (n_ts, sz, d)
tslearn-formatted dataset to be cast to stumpy format

Returns

list of arrays of shape=(d, sz_i) if d > 1 or (sz_i,) otherwise
stumpy-formatted dataset.

Examples

```
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> stumpy_arr = to_stumpy_dataset(tslearn_arr)
>>> len(stumpy_arr)
10
>>> stumpy_arr[0].shape
(16,)
>>> tslearn_arr = numpy.random.randn(10, 16, 2)
>>> stumpy_arr = to_stumpy_dataset(tslearn_arr)
>>> len(stumpy_arr)
10
>>> stumpy_arr[0].shape
(2, 16)
```

3.14.21 from_stumpy_dataset

tslearn.utils.**from_stumpy_dataset**(*X*)

Transform a stumpy-compatible dataset into a tslearn dataset.

Parameters

X: list of arrays of shapes (d, sz_i) if d > 1 or (sz_i,) otherwise
stumpy-formatted dataset.

Returns

array, shape=(n_ts, sz, d), where sz is the maximum of all array lengths
tslearn-formatted dataset

Examples

```
>>> stumpy_arr = [numpy.random.randn(10), numpy.random.randn(10)]
>>> tslearn_arr = from_stumpy_dataset(stumpy_arr)
```

(continues on next page)

(continued from previous page)

```
>>> tslearn_arr.shape
(2, 10, 1)
>>> stumpy_arr = [numpy.random.randn(3, 10), numpy.random.randn(3, 5)]
>>> tslearn_arr = from_stumpy_dataset(stumpy_arr)
>>> tslearn_arr.shape
(2, 10, 3)
```

3.14.22 to_pyflux_dataset

tslearn.utils.**to_pyflux_dataset**(*X*)

Transform a tslearn-compatible dataset into a pyflux dataset.

Parameters

X: array, shape = (n_ts, sz, d), where n_ts=1
tslearn-formatted dataset to be cast to pyflux format

Returns

Pandas data-frame
pyflux-formatted dataset (cf. link)

Notes

Conversion from/to pyflux format requires pandas to be installed.

Examples

```
>>> tslearn_arr = numpy.random.randn(1, 16, 1)
>>> pyflux_df = to_pyflux_dataset(tslearn_arr)
>>> pyflux_df.shape
(16, 1)
>>> pyflux_df.columns[0]
'dim_0'
>>> tslearn_arr = numpy.random.randn(1, 16, 2)
>>> pyflux_df = to_pyflux_dataset(tslearn_arr)
>>> pyflux_df.shape
(16, 2)
>>> pyflux_df.columns[1]
'dim_1'
>>> tslearn_arr = numpy.random.randn(10, 16, 1)
>>> to_pyflux_dataset(tslearn_arr)
Traceback (most recent call last):
...
ValueError: Array should be made of a single time series (10 here)
```

3.14.23 from_pyflux_dataset

tslearn.utils.**from_pyflux_dataset**(*X*)

Transform a pyflux-compatible dataset into a tslearn dataset.

Parameters

X: pandas data-frame
pyflux-formatted dataset

Returns

array, shape=(n_ts, sz, d), where n_ts=1
tslearn-formatted dataset. Column order is kept the same as in the original data frame.

Notes

Conversion from/to pyflux format requires pandas to be installed.

Examples

```
>>> import pandas as pd
>>> pyflux_df = pd.DataFrame()
>>> pyflux_df["dim_0"] = numpy.random.rand(10)
>>> tslearn_arr = from_pyflux_dataset(pyflux_df)
>>> tslearn_arr.shape
(1, 10, 1)
>>> pyflux_df = pd.DataFrame()
>>> pyflux_df["dim_0"] = numpy.random.rand(10)
>>> pyflux_df["dim_1"] = numpy.random.rand(10)
>>> pyflux_df["dim_2"] = numpy.random.rand(10)
>>> tslearn_arr = from_pyflux_dataset(pyflux_df)
>>> tslearn_arr.shape
(1, 10, 3)
>>> pyflux_arr = numpy.random.randn(10, 1, 16)
>>> from_pyflux_dataset(
...     pyflux_arr
... )
Traceback (most recent call last):
...
ValueError: X is not a valid input pyflux array.
```

CHAPTER

FOUR

GALLERY OF EXAMPLES

4.1 Metrics

4.2 Nearest Neighbors

4.3 Clustering and Barycenters

4.4 Classification

4.5 Automatic differentiation

4.6 Miscellaneous

4.6.1 Metrics

Canonical Time Warping

This example illustrates the use of Canonical Time Warping (CTW) between time series and plots the matches obtained by the method[1].

Note that, contrary to Dynamic Time Warping (DTW)[2], CTW can almost retrieve the ground-truth alignment (green matches) even when time series have suffered a rigid transformation (rotation+translation here).

The time series at stake in this example are color-coded trajectories whose starting (resp. end) point are depicted in blue (resp. red).

```
# Author: Romain Tavenard
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy as np

from tslearn.metrics import dtw_path, ctw_path

def plot_trajectory(ts, ax, color_code=None, alpha=1.):
    if color_code is not None:
        colors = [color_code] * len(ts)
```

(continues on next page)

[1] F. Zhou and F. Torre, "Canonical time warping for alignment of human behavior". NIPS 2009.

[2] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition". IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1), 43-49 (1978).

(continued from previous page)

```
    else:
        colors = plt.cm.jet(np.linspace(0, 1, len(ts)))
    for i in range(len(ts) - 1):
        ax.plot(ts[i:i+2, 0], ts[i:i+2, 1],
                marker='o', c=colors[i], alpha=alpha)


def get_rot2d(theta):
    return np.array(
        [[np.cos(theta), -np.sin(theta)],
         [np.sin(theta), np.cos(theta)]]
    )


def make_one_folium(sz, a=1., noise=.1, resample_fun=None):
    theta = np.linspace(0, 1, sz)
    if resample_fun is not None:
        theta = resample_fun(theta)
    theta -= .5
    theta *= .9 * np.pi
    theta = theta.reshape((-1, 1))
    r = a / 2 * (4 * np.cos(theta) - 1. / np.cos(theta))
    x = r * np.cos(theta) + np.random.rand(sz, 1) * noise
    y = r * np.sin(theta) + np.random.rand(sz, 1) * noise
    return np.array(np.hstack((x, y)))

trajectory = make_one_folium(sz=30).dot(get_rot2d(np.pi + np.pi / 3))
rotated_trajectory = trajectory.dot(get_rot2d(np.pi / 4)) + np.array([0., 3.])

path_dtw, _ = dtw_path(trajectory, rotated_trajectory)

path_ctw, cca, _ = ctw_path(trajectory, rotated_trajectory,
                            max_iter=100, n_components=2)

plt.figure(figsize=(8, 4))
ax = plt.subplot(1, 2, 1)
for (i, j) in path_dtw:
    ax.plot([trajectory[i, 0], rotated_trajectory[j, 0]],
            [trajectory[i, 1], rotated_trajectory[j, 1]],
            color='g' if i == j else 'r', alpha=.5)
plot_trajectory(trajectory, ax)
plot_trajectory(rotated_trajectory, ax)
ax.set_xticks([])
ax.set_yticks([])
ax.set_title("DTW")

ax = plt.subplot(1, 2, 2)
for (i, j) in path_ctw:
    ax.plot([trajectory[i, 0], rotated_trajectory[j, 0]],
            [trajectory[i, 1], rotated_trajectory[j, 1]],
            color='g' if i == j else 'r', alpha=.5)
plot_trajectory(trajectory, ax)
```

(continues on next page)

(continued from previous page)

```
plot_trajectory(rotated_trajectory, ax)
ax.set_xticks([])
ax.set_yticks([])
ax.set_title("CTW")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 7.153 seconds)

Dynamic Time Warping

This example illustrates Dynamic Time Warping (DTW) computation between time series and plots the optimal alignment path[1].

The image represents cost matrix, that is the squared Euclidean distance for each time point between both time series, which are represented at the left and at the top of the cost matrix.

The optimal path, that is the path that minimizes the total cost to go from the first time point to the last one, is represented in white on the image.

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt

from tslearn import metrics

numpy.random.seed(0)

s_x = numpy.array(
    [-0.790, -0.765, -0.734, -0.700, -0.668, -0.639, -0.612, -0.587, -0.564,
     -0.544, -0.529, -0.518, -0.509, -0.502, -0.494, -0.488, -0.482, -0.475,
     -0.472, -0.470, -0.465, -0.464, -0.461, -0.458, -0.459, -0.460, -0.459,
     -0.458, -0.448, -0.431, -0.408, -0.375, -0.333, -0.277, -0.196, -0.090,
     0.047, 0.220, 0.426, 0.671, 0.962, 1.300, 1.683, 2.096, 2.510, 2.895,
     3.219, 3.463, 3.621, 3.700, 3.713, 3.677, 3.606, 3.510, 3.400, 3.280,
     3.158, 3.038, 2.919, 2.801, 2.676, 2.538, 2.382, 2.206, 2.016, 1.821,
     1.627, 1.439, 1.260, 1.085, 0.917, 0.758, 0.608, 0.476, 0.361, 0.259,
     0.173, 0.096, 0.027, -0.032, -0.087, -0.137, -0.179, -0.221, -0.260,
     -0.293, -0.328, -0.359, -0.385, -0.413, -0.437, -0.458, -0.480, -0.498,
     -0.512, -0.526, -0.536, -0.544, -0.552, -0.556, -0.561, -0.565, -0.568,
     -0.570, -0.570, -0.566, -0.560, -0.549, -0.532, -0.510, -0.480, -0.443,
     -0.402, -0.357, -0.308, -0.256, -0.200, -0.139, -0.073, -0.003, 0.066,
     0.131, 0.186, 0.229, 0.259, 0.276, 0.280, 0.272, 0.256, 0.234, 0.209,
     0.186, 0.162, 0.139, 0.112, 0.081, 0.046, 0.008, -0.032, -0.071, -0.110,
     -0.147, -0.180, -0.210, -0.235, -0.256, -0.275, -0.292, -0.307, -0.320,
     -0.332, -0.344, -0.355, -0.363, -0.367, -0.364, -0.351, -0.330, -0.299,
     -0.260, -0.217, -0.172, -0.128, -0.091, -0.060, -0.036, -0.022, -0.016,
```

(continues on next page)

[1] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition". IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1), 43-49 (1978).

(continued from previous page)

```
    -0.020, -0.037, -0.065, -0.104, -0.151, -0.201, -0.253, -0.302, -0.347,
    -0.388, -0.426, -0.460, -0.491, -0.517, -0.539, -0.558, -0.575, -0.588,
    -0.600, -0.606, -0.607, -0.604, -0.598, -0.589, -0.577, -0.558, -0.531,
    -0.496, -0.454, -0.410, -0.364, -0.318, -0.276, -0.237, -0.203, -0.176,
    -0.157, -0.145, -0.142, -0.145, -0.154, -0.168, -0.185, -0.206, -0.230,
    -0.256, -0.286, -0.318, -0.351, -0.383, -0.414, -0.442, -0.467, -0.489,
    -0.508, -0.523, -0.535, -0.544, -0.552, -0.557, -0.560, -0.560, -0.557,
    -0.551, -0.542, -0.531, -0.519, -0.507, -0.494, -0.484, -0.476, -0.469,
    -0.463, -0.456, -0.449, -0.442, -0.435, -0.431, -0.429, -0.430, -0.435,
    -0.442, -0.452, -0.465, -0.479, -0.493, -0.506, -0.517, -0.526, -0.535,
    -0.548, -0.567, -0.592, -0.622, -0.655, -0.690, -0.728, -0.764, -0.795,
    -0.815, -0.823, -0.821])

s_y1 = numpy.concatenate((s_x, s_x)).reshape((-1, 1))
s_y2 = numpy.concatenate((s_x, s_x[::-1])).reshape((-1, 1))
sz = s_y1.shape[0]

path, sim = metrics.dtw_path(s_y1, s_y2)

plt.figure(figsize=(8, 8))

# definitions for the axes
left, bottom = 0.01, 0.1
w_ts = h_ts = 0.2
left_h = left + w_ts + 0.02
width = height = 0.65
bottom_h = bottom + height + 0.02

rect_s_y = [left, bottom, w_ts, height]
rect_gram = [left_h, bottom, width, height]
rect_s_x = [left_h, bottom_h, width, h_ts]

ax_gram = plt.axes(rect_gram)
ax_s_x = plt.axes(rect_s_x)
ax_s_y = plt.axes(rect_s_y)

mat = cdist(s_y1, s_y2)

ax_gram.imshow(mat, origin='lower')
ax_gram.axis("off")
ax_gram.autoscale(False)
ax_gram.plot([j for (i, j) in path], [i for (i, j) in path], "w-",
             linewidth=3.)

ax_s_x.plot(numpy.arange(sz), s_y2, "b-", linewidth=3.)
ax_s_x.axis("off")
ax_s_x.set_xlim((0, sz - 1))

ax_s_y.plot(- s_y1, numpy.arange(sz), "b-", linewidth=3.)
ax_s_y.axis("off")
ax_s_y.set_ylim((0, sz - 1))
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 0.154 seconds)

DTW computation with a custom distance metric

This example illustrates how to use the DTW computation of the optimal alignment path[1] on a user-defined distance matrix using *dtw_path_from_metric()*.

Left is the DTW of two angular time series using the length of the arc on the unit circle as a distance metric[2] and right is the DTW of two multidimensional boolean time series using hamming distance[3].

The images represent cost matrices, that is, on the left the length of the arc between each pair of angles on the unit circle and on the right the hamming distances between the multidimensional boolean arrays. In both cases, the corresponding time series are represented at the left and at the top of each cost matrix.

The optimal path, that is the path that minimizes the total user-defined cost from the first time point to the last one, is represented in white on the image.

-
-

```
# Author: Romain Fayat
# License: BSD 3 clause
# sphinx_gallery_thumbnail_number = 2

import numpy as np
from numpy import pi
from sklearn.metrics import pairwise_distances
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap

from tslearn.generators import random_walks
from tslearn import metrics
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

np.random.seed(0)
n_ts, sz = 2, 100


# Example 1 : Length of the arc between two angles on a circle
def arc_length(angle_1, angle_2, r=1.):
    """Length of the arc between two angles (in rad) on a circle of
    radius r.
    """
    # Compute the angle between the two inputs between 0 and 2*pi.
    theta = np.mod(angle_2 - angle_1, 2*pi)
    if theta > pi:
        theta = theta - 2 * pi
    # Return the length of the arc
```

(continues on next page)

[1] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition". IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1), 43-49 (1978).

[2] Definition of the length of an arc on Wikipedia.

[3] See Hammig distance in Scipy's documentation.

(continued from previous page)

```
    L = r * np.abs(theta)
    return L[0]


dataset_1 = random_walks(n_ts=n_ts, sz=sz, d=1)
scaler = TimeSeriesScalerMeanVariance(mu=0., std=pi)  # Rescale the time series
dataset_scaled_1 = scaler.fit_transform(dataset_1)

# DTW using a function as the metric argument
path_1, sim_1 = metrics.dtw_path_from_metric(
    dataset_scaled_1[0], dataset_scaled_1[1], metric=arc_length
)

# Example 2 : Hamming distance between 2 multi-dimensional boolean time series
rw = random_walks(n_ts=n_ts, sz=sz, d=15, std=.3)
dataset_2 = np.mod(np.floor(rw), 4) == 0

# DTW using one of the options of sklearn.metrics.pairwise_distances
path_2, sim_2 = metrics.dtw_path_from_metric(
    dataset_2[0], dataset_2[1], metric="hamming"
)

# Plots
# Compute the distance matrices for the plots
distances_1 = pairwise_distances(
    dataset_scaled_1[0], dataset_scaled_1[1], metric=arc_length
)
distances_2 = pairwise_distances(dataset_2[0], dataset_2[1], metric="hamming")

# Definitions for the axes
left, bottom = 0.01, 0.1
w_ts = h_ts = 0.2
left_h = left + w_ts + 0.02
width = height = 0.65
bottom_h = bottom + height + 0.02

rect_s_y = [left, bottom, w_ts, height]
rect_dist = [left_h, bottom, width, height]
rect_s_x = [left_h, bottom_h, width, h_ts]

# Plot example 1
plt.figure(1, figsize=(6, 6))
ax_dist = plt.axes(rect_dist)
ax_s_x = plt.axes(rect_s_x)
ax_s_y = plt.axes(rect_s_y)

ax_dist.imshow(distances_1, origin='lower')
ax_dist.axis("off")
ax_dist.autoscale(False)
ax_dist.plot(*zip(*path_1), "w-", linewidth=3.)

ticks_location = [-pi, 0, pi]
```

(continues on next page)

(continued from previous page)

```
ticks_labels = [r"$\bf-\pi$", r"$\bf0$", r"$\bf\pi$"]

ax_s_x.plot([0, sz - 1], [ticks_location]*2, "k--", alpha=.2)
ax_s_x.plot(np.arange(sz), dataset_scaled_1[1], "b-", linewidth=3.)
ax_s_x.set_xlim((0, sz - 1))
ax_s_x.axis("off")

ax_s_y.plot([ticks_location]*2, [0, sz - 1], "k--", alpha=.2)
ax_s_y.plot(-dataset_scaled_1[0], np.arange(sz), "b-", linewidth=3.)
ax_s_y.set_ylim((0, sz - 1))
ax_s_y.axis("off")

for loc, s in zip(ticks_location, ticks_labels):
    ax_s_x.text(0, loc, s, fontsize="large", color="grey",
                horizontalalignment="right", verticalalignment="center")
    ax_s_y.text(-loc, 0, s, fontsize="large", color="grey",
                horizontalalignment="center", verticalalignment="top")

# Plot example 2
plt.figure(2, figsize=(6, 6))
ax_dist = plt.axes(rect_dist)
ax_s_x = plt.axes(rect_s_x)
ax_s_y = plt.axes(rect_s_y)

ax_dist.imshow(distances_2, origin='lower')
ax_dist.axis("off")
ax_dist.autoscale(False)
ax_dist.plot(*zip(*path_2), "w-", linewidth=3.)

colors = [(1, 1, 1), (0, 0, 1)]  # White -> Blue
cmap_name = 'white_blue'
cm = LinearSegmentedColormap.from_list(cmap_name, colors, N=2)

ax_s_x.imshow(dataset_2[1].T, aspect="auto", cmap=cm)
ax_s_x.axis("off")

ax_s_y.imshow(np.flip(dataset_2[0], axis=1), aspect="auto", cmap=cm)
ax_s_y.axis("off")

plt.show()
```

Total running time of the script: (0 minutes 0.567 seconds)

Frechet

This example illustrates the use of Frechet distance between time series and plots the matches obtained by the method[1] compared to DTW.

The Frechet distance is plotted in red:

$$Frechet(X, Y) = \max_{(i,j)\in\pi} \|X_i - Y_j\|$$

[1] FRÉCHET, M. "Sur quelques points du calcul fonctionnel. Rendiconti del Circolo Mathematico di Palermo", 22, 1–74, 1906.

```
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy as np

from tslearn.metrics import frechet_path, dtw_path

np.random.seed(42)

nb_points = 100
angle1 = 0.25*np.linspace(0, 4*np.pi, nb_points)
s1 = np.sin(angle1) + 0.1 * np.random.rand(nb_points) + 1
angle2 = np.linspace(0, 2 * np.pi, nb_points)
s2 = 0.5 * np.sin(angle2) + 0.1 * np.random.rand(nb_points)

path_dtw, _ = dtw_path(s1, s2)
path_frechet, distance_frechet = frechet_path(s1, s2)

plt.figure(figsize=(8, 4))
ax = plt.subplot(1, 2, 1)
ax.plot(s1)
ax.plot(s2)
for (i, j) in path_frechet:
    is_max = np.linalg.norm(s1[i] - s2[j]) == distance_frechet
    ax.plot(
        [i, j],
        [s1[i], s2[j]],
        'rd:' if is_max else 'k--',
        alpha=1 if is_max else 0.1
    )
ax.set_title("Frechet")

ax = plt.subplot(1, 2, 2)
ax.plot(s1)
ax.plot(s2)
for (i, j) in path_dtw:
    ax.plot([i, j],[s1[i], s2[j]], 'k--', alpha=0.1)
ax.set_title("DTW")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 1.281 seconds)

LB_Keogh

This example illustrates the principle of time series envelope and its relationship to the "LB_Keogh" lower bound[1].

The envelope of a time series consists of two time series such that the original time series is between the two time series. Denoting the original time series $X = (X_i)_{1 \leq i \leq n}$, the envelope of this time series is an ensemble of two time

[1] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping". Knowledge and Information Systems, 7(3), 358-386 (2004).

series of same length $L = (l_i)_{1 \leq i \leq n}$ and $U = (u_i)_{1 \leq i \leq n}$ such that for all $i \in \{1, \ldots, n\}$:

$$u_i = \max(x_{i-r}, \ldots, x_{i+r})$$
$$l_i = \min(x_{i-r}, \ldots, x_{i+r})$$

where r is the radius of the envelope.

The distance between a time series Q and an envelope (L, U) is defined as:

$$LB_{Keogh}(Q, (L, U)) = \sqrt{\sum_{i=1}^{n} \begin{cases} (q_i - u_i)^2 & \text{if } q_i > u_i \\ (q_i - l_i)^2 & \text{if } q_i < l_i \\ 0 & \text{otherwise} \end{cases}}$$

So it is simply the Euclidean distance between Q and the envelope.

-
-

```
# Author: Romain Tavenard
#         Johann Faouzi
# License: BSD 3 clause
# sphinx_gallery_thumbnail_number = 2

import numpy
import matplotlib.pyplot as plt

from tslearn.generators import random_walks
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn import metrics

numpy.random.seed(0)
n_ts, sz, d = 2, 100, 1
dataset = random_walks(n_ts=n_ts, sz=sz, d=d)
scaler = TimeSeriesScalerMeanVariance(mu=0., std=1.)  # Rescale time series
dataset_scaled = scaler.fit_transform(dataset)

plt.figure(figsize=(14, 8))
envelope_down, envelope_up = metrics.lb_envelope(dataset_scaled[0], radius=3)
plt.plot(dataset_scaled[0, :, 0], "r-", label='First time series')
plt.plot(envelope_down[:, 0], "b-", label='Lower envelope')
plt.plot(envelope_up[:, 0], "g-", label='Upper envelope')
plt.legend()
plt.title('Envelope around a time series with radius=3')

plt.figure(figsize=(14, 8))
plt.plot(envelope_down[:, 0], "b-", label='Lower envelope')
plt.plot(envelope_up[:, 0], "g-", label='Upper envelope')
plt.plot(dataset_scaled[1, :, 0], "k-", label='Second time series')
plt.vlines(numpy.arange(sz), dataset_scaled[1, :, 0], numpy.clip(
    dataset_scaled[1, :, 0], envelope_down[:, 0], envelope_up[:, 0]),
           label='Distance', color='orange')
plt.legend()
lb_k_sim = metrics.lb_keogh(dataset_scaled[1],
                            envelope_candidate=(envelope_down, envelope_up))
```

(continues on next page)

(continued from previous page)

```
plt.title('Distance between the second time series and \n'
          'the envelope = {:.4f}'.format(lb_k_sim))

plt.show()
```

Total running time of the script: (0 minutes 0.880 seconds)

Longest Common Subsequence

This example illustrates LCSS computation between time series and plots the alignment path[1]. and its relationship to the DTW.

Since LCSS focuses on the similar parts between two time-series, a potential use case is to identify the similarity between time-series whose lengths differ greatly or have noise. As one example, M. Vlachos et al.[1] used this method to cluster time series regarding human writing in the presence of noise.

The example demonstrates the use of the functions *lcss_path* and *dtw_path* to calculate the alignment path between them and compare the two approaches when dealing with unequal-length sequence data and noise.

-
-

```
# Author: Daniela Duarte
# 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, random_state=5)
scaler = TimeSeriesScalerMeanVariance(mu=0., std=1.)  # Rescale time series
dataset_scaled = scaler.fit_transform(dataset)

lcss_path, sim_lcss = metrics.lcss_path(dataset_scaled[0, :, 0], dataset_scaled[1, :40,␣
↪0], eps=1.5)
dtw_path, sim_dtw = metrics.dtw_path(dataset_scaled[0, :, 0], dataset_scaled[1, :40, 0])

plt.figure(1, figsize=(8, 8))

plt.plot(dataset_scaled[0, :, 0], "b-", label='First time series')
plt.plot(dataset_scaled[1, :40, 0], "g-", label='Second time series')

for positions in lcss_path:
    plt.plot([positions[0], positions[1]],
             [dataset_scaled[0, positions[0], 0], dataset_scaled[1, positions[1], 0]],␣
```

(continues on next page)

[1] M. Vlachos, D. Gunopoulos, and G. Kollios. 2002. "Discovering Similar Multidimensional Trajectories", In Proceedings of the 18th International Conference on Data Engineering (ICDE '02). IEEE Computer Society, USA, 673.

(continued from previous page)

```
↪color='orange')
plt.legend()
plt.title("Time series matching with LCSS")

plt.figure(2, figsize=(8, 8))
plt.plot(dataset_scaled[0, :, 0], "b-", label='First time series')
plt.plot(dataset_scaled[1, :40, 0], "g-", label='Second time series')

for positions in dtw_path:
    plt.plot([positions[0], positions[1]],
             [dataset_scaled[0, positions[0], 0], dataset_scaled[1, positions[1], 0]],␣
↪color='orange')

plt.legend()
plt.title("Time series matching with DTW")

plt.show()
```

Total running time of the script: (0 minutes 0.563 seconds)

Longest Common Subsequence with a custom distance metric

This example illustrates how to use the LCSS computation of the alignment path[1] on a user-defined distance matrix using *dtw_path_from_metric()*.

The example is the LCSS of two angular time series using the length of the arc on the unit circle as a distance metric[2].

The image represent cost matrices, that is, the length of the arc between each pair of angles on the unit circle. The corresponding time series are represented at the left and at the top of each cost matrix.

The alignment path, that is the path where the matches between the two time-series occurred within the pre-defined threshold, is represented in white on the image.

```
# Author: Daniela Duarte
# License: BSD 3 clause

import numpy as np
from numpy import pi
from sklearn.metrics import pairwise_distances
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap

from tslearn.generators import random_walks
from tslearn import metrics
from tslearn.preprocessing import TimeSeriesScalerMeanVariance


np.random.seed(0)
n_ts, sz = 2, 100
```

(continues on next page)

[1] M. Vlachos, D. Gunopoulos, and G. Kollios. 2002. "Discovering Similar Multidimensional Trajectories", In Proceedings of the 18th International Conference on Data Engineering (ICDE '02). IEEE Computer Society, USA, 673.

[2] Definition of the length of an arc on Wikipedia.

(continued from previous page)

```
# Example : Length of the arc between two angles on a circle
def arc_length(angle_1, angle_2, r=1.):
    """Length of the arc between two angles (in rad) on a circle of
    radius r.
    """
    # Compute the angle between the two inputs between 0 and 2*pi.
    theta = np.mod(angle_2 - angle_1, 2*pi)
    if theta > pi:
        theta = theta - 2 * pi
    # Return the length of the arc
    L = r * np.abs(theta)
    return L[0]


dataset_1 = random_walks(n_ts=n_ts, sz=sz, d=1)
scaler = TimeSeriesScalerMeanVariance(mu=0., std=pi)  # Rescale the time series
dataset_scaled_1 = scaler.fit_transform(dataset_1)

# LCSS using a function as the metric argument
path_1, sim_1 = metrics.lcss_path_from_metric(
    dataset_scaled_1[0], dataset_scaled_1[1], eps=1, metric=arc_length
)

# Plots
# Compute the distance matrices for the plot
distances_1 = pairwise_distances(
    dataset_scaled_1[0], dataset_scaled_1[1], metric=arc_length
)

# Definitions for the axes
left, bottom = 0.01, 0.1
w_ts = h_ts = 0.2
left_h = left + w_ts + 0.02
width = height = 0.65
bottom_h = bottom + height + 0.02

rect_s_y = [left, bottom, w_ts, height]
rect_dist = [left_h, bottom, width, height]
rect_s_x = [left_h, bottom_h, width, h_ts]

# Plot example
plt.figure(figsize=(6, 6))
ax_dist = plt.axes(rect_dist)
ax_s_x = plt.axes(rect_s_x)
ax_s_y = plt.axes(rect_s_y)

ax_dist.imshow(distances_1, origin='lower')
ax_dist.axis("off")
ax_dist.autoscale(False)
ax_dist.plot(*zip(*path_1), "w-", linewidth=3.)
```

(continues on next page)

(continued from previous page)

```
ticks_location = [-pi, 0, pi]
ticks_labels = [r"$\bf-\pi$", r"$\bf0$", r"$\bf\pi$"]

ax_s_x.plot([0, sz - 1], [ticks_location]*2, "k--", alpha=.2)
ax_s_x.plot(np.arange(sz), dataset_scaled_1[1], "b-", linewidth=3.)
ax_s_x.set_xlim((0, sz - 1))
ax_s_x.axis("off")

ax_s_y.plot([ticks_location]*2, [0, sz - 1], "k--", alpha=.2)
ax_s_y.plot(-dataset_scaled_1[0], np.arange(sz), "b-", linewidth=3.)
ax_s_y.set_ylim((0, sz - 1))
ax_s_y.axis("off")

for loc, s in zip(ticks_location, ticks_labels):
    ax_s_x.text(0, loc, s, fontsize="large", color="grey",
                horizontalalignment="right", verticalalignment="center")
    ax_s_y.text(-loc, 0, s, fontsize="large", color="grey",
                horizontalalignment="center", verticalalignment="top")

plt.show()
```

Total running time of the script: (0 minutes 0.595 seconds)

sDTW multi path matching

This example illustrates how subsequent DTW can be used to find multiple matches of a sequence in a longer sequence.

A potential usecase is to identify the occurrence of certain events in continuous sensor signals. As one example Barth et al.[1] used this method to find stride in sensor recordings of gait.

The example demonstrates the use of the functions *subsequence_cost_matrix* and *subsequence_path* to manually calculate warping paths from multiple potential alignments. If you are only interested in finding the optimal alignment, you can directly use *dtw_subsequence_path*.

```
Shape long sequence: (500, 1)
Shape short sequence: (100, 1)
```

```
# Author: Arne Kuederle
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy
from scipy.signal import find_peaks

from tslearn import metrics
from tslearn.generators import random_walks
```

(continues on next page)

[1] Barth, et al. (2013): Subsequence dynamic time warping as a method for robust step segmentation using gyroscope signals of daily life activities, EMBS, https://doi.org/10.1109/EMBC.2013.6611104

(continued from previous page)

```
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

numpy.random.seed(0)
n_ts, sz, d = 2, 100, 1
n_repeat = 5
dataset = random_walks(n_ts=n_ts, sz=sz, d=d)
scaler = TimeSeriesScalerMeanVariance(mu=0., std=1.)  # Rescale time series
dataset_scaled = scaler.fit_transform(dataset)

# We repeat the long sequence multiple times to generate multiple possible
# matches
long_sequence = numpy.tile(dataset_scaled[1], (n_repeat, 1))
short_sequence = dataset_scaled[0]

sz1 = len(long_sequence)
sz2 = len(short_sequence)

print('Shape long sequence: {}'.format(long_sequence.shape))
print('Shape short sequence: {}'.format(short_sequence.shape))

# Calculate the accumulated cost matrix
mat = metrics.subsequence_cost_matrix(short_sequence,
                                      long_sequence)

# Calculate cost function
cost_func = mat[-1, :]

# Identify potential matches in the cost function (parameters are tuned to
# fit this example)
potential_matches = find_peaks(-cost_func, distance=sz * 0.75, height=-50)[0]

# Calculate the optimal warping path starting from each of the identified
# minima
paths = [metrics.subsequence_path(mat, match) for match in
         potential_matches]

plt.figure(1, figsize=(6 * n_repeat, 6))

# definitions for the axes
left, bottom = 0.01, 0.1
h_ts = 0.2
w_ts = h_ts / n_repeat
left_h = left + w_ts + 0.02
width = height = 0.65
bottom_h = bottom + height + 0.02

rect_s_y = [left, bottom, w_ts, height]
rect_gram = [left_h, bottom, width, height]
rect_s_x = [left_h, bottom_h, width, h_ts]

ax_gram = plt.axes(rect_gram)
ax_s_x = plt.axes(rect_s_x)
```

(continues on next page)

(continued from previous page)

```
ax_s_y = plt.axes(rect_s_y)

ax_gram.imshow(numpy.sqrt(mat))
ax_gram.axis("off")
ax_gram.autoscale(False)

# Plot the paths
for path in paths:
    ax_gram.plot([j for (i, j) in path], [i for (i, j) in path], "w-",
                 linewidth=3.)

ax_s_x.plot(numpy.arange(sz1), long_sequence, "b-", linewidth=3.)
ax_s_x.axis("off")
ax_s_x.set_xlim((0, sz1 - 1))

ax_s_y.plot(- short_sequence, numpy.arange(sz2)[::-1], "b-", linewidth=3.)
ax_s_y.axis("off")
ax_s_y.set_ylim((0, sz2 - 1))

plt.show()
```

Total running time of the script: (0 minutes 0.692 seconds)

Soft Dynamic Time Warping

This example illustrates Soft Dynamic Time Warping (DTW) computation between time series and plots the optimal soft alignment matrices[1].

-
-
-

```
# Author: Romain Tavenard
# License: BSD 3 clause
# sphinx_gallery_thumbnail_number = 3

import numpy
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt

from tslearn import metrics

numpy.random.seed(0)

s_x = numpy.array(
    [-0.790, -0.765, -0.734, -0.700, -0.668, -0.639, -0.612, -0.587, -0.564,
     -0.544, -0.529, -0.518, -0.509, -0.502, -0.494, -0.488, -0.482, -0.475,
     -0.472, -0.470, -0.465, -0.464, -0.461, -0.458, -0.459, -0.460, -0.459,
     -0.458, -0.448, -0.431, -0.408, -0.375, -0.333, -0.277, -0.196, -0.090,
     0.047, 0.220, 0.426, 0.671, 0.962, 1.300, 1.683, 2.096, 2.510, 2.895,
```

(continues on next page)

[1] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.

(continued from previous page)

```
    3.219, 3.463, 3.621, 3.700, 3.713, 3.677, 3.606, 3.510, 3.400, 3.280,
    3.158, 3.038, 2.919, 2.801, 2.676, 2.538, 2.382, 2.206, 2.016, 1.821,
    1.627, 1.439, 1.260, 1.085, 0.917, 0.758, 0.608, 0.476, 0.361, 0.259,
    0.173, 0.096, 0.027, -0.032, -0.087, -0.137, -0.179, -0.221, -0.260,
    -0.293, -0.328, -0.359, -0.385, -0.413, -0.437, -0.458, -0.480, -0.498,
    -0.512, -0.526, -0.536, -0.544, -0.552, -0.556, -0.561, -0.565, -0.568,
    -0.570, -0.570, -0.566, -0.560, -0.549, -0.532, -0.510, -0.480, -0.443,
    -0.402, -0.357, -0.308, -0.256, -0.200, -0.139, -0.073, -0.003, 0.066,
    0.131, 0.186, 0.229, 0.259, 0.276, 0.280, 0.272, 0.256, 0.234, 0.209,
    0.186, 0.162, 0.139, 0.112, 0.081, 0.046, 0.008, -0.032, -0.071, -0.110,
    -0.147, -0.180, -0.210, -0.235, -0.256, -0.275, -0.292, -0.307, -0.320,
    -0.332, -0.344, -0.355, -0.363, -0.367, -0.364, -0.351, -0.330, -0.299,
    -0.260, -0.217, -0.172, -0.128, -0.091, -0.060, -0.036, -0.022, -0.016,
    -0.020, -0.037, -0.065, -0.104, -0.151, -0.201, -0.253, -0.302, -0.347,
    -0.388, -0.426, -0.460, -0.491, -0.517, -0.539, -0.558, -0.575, -0.588,
    -0.600, -0.606, -0.607, -0.604, -0.598, -0.589, -0.577, -0.558, -0.531,
    -0.496, -0.454, -0.410, -0.364, -0.318, -0.276, -0.237, -0.203, -0.176,
    -0.157, -0.145, -0.142, -0.145, -0.154, -0.168, -0.185, -0.206, -0.230,
    -0.256, -0.286, -0.318, -0.351, -0.383, -0.414, -0.442, -0.467, -0.489,
    -0.508, -0.523, -0.535, -0.544, -0.552, -0.557, -0.560, -0.560, -0.557,
    -0.551, -0.542, -0.531, -0.519, -0.507, -0.494, -0.484, -0.476, -0.469,
    -0.463, -0.456, -0.449, -0.442, -0.435, -0.431, -0.429, -0.430, -0.435,
    -0.442, -0.452, -0.465, -0.479, -0.493, -0.506, -0.517, -0.526, -0.535,
    -0.548, -0.567, -0.592, -0.622, -0.655, -0.690, -0.728, -0.764, -0.795,
    -0.815, -0.823, -0.821])

s_y1 = numpy.concatenate((s_x, s_x))[::2].reshape((-1, 1))
s_y2 = numpy.concatenate((s_x, s_x[::-1]))[::2].reshape((-1, 1))
sz = s_y1.shape[0]

for gamma in [0., .1, 1.]:
    alignment, sim = metrics.soft_dtw_alignment(s_y1, s_y2, gamma=gamma)

    plt.figure(figsize=(8, 8))

    # definitions for the axes
    left, bottom = 0.01, 0.1
    w_ts = h_ts = 0.2
    left_h = left + w_ts + 0.02
    width = height = 0.65
    bottom_h = bottom + height + 0.02

    rect_s_y = [left, bottom, w_ts, height]
    rect_gram = [left_h, bottom, width, height]
    rect_s_x = [left_h, bottom_h, width, h_ts]

    ax_gram = plt.axes(rect_gram)
    ax_s_x = plt.axes(rect_s_x)
    ax_s_y = plt.axes(rect_s_y)

    mat = cdist(s_y1, s_y2)
```

(continues on next page)

(continued from previous page)

```
    ax_gram.imshow(alignment, origin='lower')
    ax_gram.axis("off")
    ax_gram.autoscale(False)
    plt.suptitle("$\\gamma={:.1f}$".format(gamma), fontsize=24)

    ax_s_x.plot(numpy.arange(sz), s_y2, "b-", linewidth=3.)
    ax_s_x.axis("off")
    ax_s_x.set_xlim((0, sz - 1))

    ax_s_y.plot(- s_y1, numpy.arange(sz), "b-", linewidth=3.)
    ax_s_y.axis("off")
    ax_s_y.set_ylim((0, sz - 1))

    plt.show()
```

Total running time of the script: (0 minutes 0.727 seconds)

4.6.2 Nearest Neighbors

k-NN search

This example performs a k-Nearest-Neighbor search in a database of time series using DTW as a base metric.

To do so, we use the *tslearn.neighbors.KNeighborsTimeSeries* class which provides utilities for the k-Nearest-Neighbor algorithm for time series.

[1] Wikipedia entry for the k-nearest neighbors algorithm

[2] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition". IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1), 43-49 (1978).

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.neighbors import KNeighborsTimeSeries
from tslearn.datasets import CachedDatasets

seed = 0
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)
```

(continues on next page)

(continued from previous page)

```
    plt.plot(X_test[idx_ts].ravel(), "k-")
    plt.xticks([])
    for rank_nn in range(n_neighbors):
        plt.subplot(n_neighbors + 1, n_queries,
                    idx_ts + (n_queries * (rank_nn + 1)) + 1)
        plt.plot(X_train[ind[idx_ts, rank_nn]].ravel(), "r-")
        plt.xticks([])


plt.suptitle("Queries (in black) and their nearest neighbors (red)")
plt.show()
```

Total running time of the script: (0 minutes 0.915 seconds)

Hyper-parameter tuning of a pipeline with KNeighbors time series classifier

In this example, we demonstrate how it is possible to use the different algorithms of tslearn in combination with sklearn utilities, such as the *sklearn.pipeline.Pipeline* and *sklearn.model_selection.GridSearchCV*. In this specific example, we will tune two of the hyper-parameters of a *KNeighborsTimeSeriesClassifier*.

```
Performing hyper-parameter tuning of KNN classifier... Done!

Got the following accuracies on the test set for each fold:
|n_neighbors |  weights   |score_fold_1|score_fold_2|score_fold_3|
-----------------------------------------------------------------
|           5|     uniform|     0.64706|     0.82353|      0.6875|
|           5|    distance|     0.70588|     0.88235|      0.8125|
|          25|     uniform|     0.64706|     0.64706|       0.625|
|          25|    distance|     0.82353|     0.76471|      0.8125|

Best parameter combination:
weights=distance, n_neighbors=5
```

```
# Author: Gilles Vandewiele
# License: BSD 3 clause

from tslearn.neighbors import KNeighborsTimeSeriesClassifier
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.datasets import CachedDatasets

from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.pipeline import Pipeline

import numpy as np

import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
# Our pipeline consists of two phases. First, data will be normalized using
# min-max normalization. Afterwards, it is fed to a KNN classifier. For the
# KNN classifier, we tune the n_neighbors and weights hyper-parameters.
n_splits = 3
pipeline = GridSearchCV(
    Pipeline([
            ('normalize', TimeSeriesScalerMinMax()),
            ('knn', KNeighborsTimeSeriesClassifier())
    ]),
    {'knn__n_neighbors': [5, 25], 'knn__weights': ['uniform', 'distance']},
    cv=StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)
)

X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Keep only timeseries of class 1, 2, 3
X_train = X_train[y_train > 0]
y_train = y_train[y_train > 0]

# Keep only the first 50 timeseries of both train and
# retain only a small amount of each of the timeseries
X_train, y_train = X_train[:50, 50:150], y_train[:50]

# Plot our timeseries
colors = ['g', 'b', 'r']
plt.figure()
for ts, label in zip(X_train, y_train):
    plt.plot(ts, c=colors[label - 2], alpha=0.5)
plt.title('The timeseries in the dataset')
plt.tight_layout()
plt.show()

# Fit our pipeline
print(end='Performing hyper-parameter tuning of KNN classifier... ')
pipeline.fit(X_train, y_train)
results = pipeline.cv_results_

# Print each possible configuration parameter and the out-of-fold accuracies
print('Done!')
print()
print('Got the following accuracies on the test set for each fold:')

header_str = '|'
columns = ['n_neighbors', 'weights']
columns += ['score_fold_{}'.format(i + 1) for i in range(n_splits)]
for col in columns:
    header_str += '{:^12}|'.format(col)
print(header_str)
print('-'*(len(columns) * 13))

for i in range(len(results['params'])):
    s = '|'
```

(continues on next page)

(continued from previous page)

```
    s += '{:>12}|'.format(results['params'][i]['knn__n_neighbors'])
    s += '{:>12}|'.format(results['params'][i]['knn__weights'])
    for k in range(n_splits):
        score = results['split{}_test_score'.format(k)][i]
        score = np.around(score, 5)
        s += '{:>12}|'.format(score)
    print(s.strip())

best_comb = np.argmax(results['mean_test_score'])
best_params = results['params'][best_comb]

print()
print('Best parameter combination:')
print('weights={}, n_neighbors={}'.format(best_params['knn__weights'],
                                          best_params['knn__n_neighbors']))
```

Total running time of the script: (0 minutes 0.556 seconds)

Nearest neighbors

This example illustrates the use of nearest neighbor methods for database search and classification tasks.

The three-nearest neighbors of the time series from a test set are computed. Then, the predictive performance of a three-nearest neighbors classifier [1] is computed with three different metrics: Dynamic Time Warping [2], Euclidean distance and SAX-MINDIST [3].

[1] Wikipedia entry for the k-nearest neighbors algorithm

[2] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition". IEEE Transactions on Acoustics, Speech, and Signal Processing, 26(1), 43-49 (1978).

[3] J. Lin, E. Keogh, L. Wei and S. Lonardi, "Experiencing SAX: a novel symbolic representation of time series". Data Mining and Knowledge Discovery, 15(2), 107-144 (2007).

```
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]
```

(continues on next page)

(continued from previous page)

```
 [18 10  2]
 [10 12  2]]
First nearest neighbor class: [0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 0 1 0 0 0]

2. Nearest neighbor classification using DTW
Correct classification rate: 1.0

3. Nearest neighbor classification using L2
Correct classification rate: 1.0

4. Nearest neighbor classification using SAX+MINDIST
Correct classification rate: 0.5
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
from sklearn.metrics import accuracy_score

from tslearn.generators import random_walk_blobs
from tslearn.preprocessing import TimeSeriesScalerMinMax, \
    TimeSeriesScalerMeanVariance
from tslearn.neighbors import KNeighborsTimeSeriesClassifier, \
    KNeighborsTimeSeries

numpy.random.seed(0)
n_ts_per_blob, sz, d, n_blobs = 20, 100, 1, 2

# Prepare data
X, y = random_walk_blobs(n_ts_per_blob=n_ts_per_blob,
                         sz=sz,
                         d=d,
                         n_blobs=n_blobs)
scaler = TimeSeriesScalerMinMax(value_range=(0., 1.))  # Rescale time series
X_scaled = scaler.fit_transform(X)

indices_shuffle = numpy.random.permutation(n_ts_per_blob * n_blobs)
X_shuffle = X_scaled[indices_shuffle]
y_shuffle = y[indices_shuffle]

X_train = X_shuffle[:n_ts_per_blob * n_blobs // 2]
X_test = X_shuffle[n_ts_per_blob * n_blobs // 2:]
y_train = y_shuffle[:n_ts_per_blob * n_blobs // 2]
y_test = y_shuffle[n_ts_per_blob * n_blobs // 2:]

# Nearest neighbor search
knn = KNeighborsTimeSeries(n_neighbors=3, metric="dtw")
```

(continues on next page)

(continued from previous page)

```
knn.fit(X_train, y_train)
dists, ind = knn.kneighbors(X_test)
print("1. Nearest neighbour search")
print("Computed nearest neighbor indices (wrt DTW)\n", ind)
print("First nearest neighbor class:", y_test[ind[:, 0]])

# Nearest neighbor classification
knn_clf = KNeighborsTimeSeriesClassifier(n_neighbors=3, metric="dtw")
knn_clf.fit(X_train, y_train)
predicted_labels = knn_clf.predict(X_test)
print("\n2. Nearest neighbor classification using DTW")
print("Correct classification rate:", accuracy_score(y_test, predicted_labels))

# Nearest neighbor classification with a different metric (Euclidean distance)
knn_clf = KNeighborsTimeSeriesClassifier(n_neighbors=3, metric="euclidean")
knn_clf.fit(X_train, y_train)
predicted_labels = knn_clf.predict(X_test)
print("\n3. Nearest neighbor classification using L2")
print("Correct classification rate:", accuracy_score(y_test, predicted_labels))

# Nearest neighbor classification based on SAX representation
metric_params = {'n_segments': 10, 'alphabet_size_avg': 5}
knn_clf = KNeighborsTimeSeriesClassifier(n_neighbors=3, metric="sax",
                                         metric_params=metric_params)
knn_clf.fit(X_train, y_train)
predicted_labels = knn_clf.predict(X_test)
print("\n4. Nearest neighbor classification using SAX+MINDIST")
print("Correct classification rate:", accuracy_score(y_test, predicted_labels))
```

Total running time of the script: (0 minutes 0.879 seconds)

1-NN with SAX + MINDIST

This example presents a comparison between k-Nearest Neighbor runs with k=1. It compares the use of: * MINDIST (see [1]) on SAX representations of the data. * Euclidean distance on the raw values of the time series.

The comparison is based on test accuracy using several benchmark datasets.

[1] Lin, Jessica, et al. "Experiencing SAX: a novel symbolic
representation of time series." Data Mining and knowledge discovery 15.2 (2007): 107-144.

dataset	sax error	sax time	eucl error	eucl time
SyntheticControl	0.03	1.91086	0.12	0.01995
GunPoint	0.20667	0.40425	0.08667	0.01
FaceFour	0.14773	0.36558	0.21591	0.00775
Lightning2	0.19672	0.72261	0.2459	0.00867
Lightning7	0.46575	0.48113	0.42466	0.00893
ECG200	0.12	0.36926	0.12	0.00984
Plane	0.04762	0.4775	0.0381	0.01124
Car	0.31667	0.66417	0.26667	0.0084
Beef	0.5	0.23669	0.33333	0.00607
Coffee	0.46429	0.15693	0.0	0.00593

(continues on next page)

(continued from previous page)

```
|            OliveOil|     0.83333|     0.31684|     0.13333|     0.00619|
--------------------------------------------------------------------------------
```

```
# Author: Gilles Vandewiele
# License: BSD 3 clause

import time
import warnings

import numpy

from sklearn.base import clone
from sklearn.metrics import accuracy_score

from tslearn.datasets import UCR_UEA_datasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn.neighbors import KNeighborsTimeSeriesClassifier


warnings.filterwarnings('ignore')


def print_table(accuracies, times):
    """Utility function to pretty print the obtained accuracies"""
    header_str = '|'
    header_str += '{:^20}|'.format('dataset')
    columns = ['sax error', 'sax time', 'eucl error', 'eucl time']
    for col in columns:
        header_str += '{:^12}|'.format(col)
    print(header_str)
    print('-'*(len(columns) * 13 + 22))

    for dataset in accuracies:
        acc_sax, acc_euclidean = accuracies[dataset]
        time_sax, time_euclidean = times[dataset]
        sax_error = numpy.around(1 - acc_sax, 5)
        eucl_error = numpy.around(1 - acc_euclidean, 5)
        time_sax = numpy.around(time_sax, 5)
        time_euclidean = numpy.around(time_euclidean, 5)
        s = '|'
        s += '{:>20}|'.format(dataset)
        s += '{:>12}|'.format(sax_error)
        s += '{:>12}|'.format(time_sax)
        s += '{:>12}|'.format(eucl_error)
        s += '{:>12}|'.format(time_euclidean)
        print(s.strip())
```

(continues on next page)

(continued from previous page)

```
    print('-'*(len(columns) * 13 + 22))


# Set seed
numpy.random.seed(0)

# Defining dataset and the number of segments
data_loader = UCR_UEA_datasets()
datasets = [
    ('SyntheticControl', 16),
    ('GunPoint', 64),
    ('FaceFour', 128),
    ('Lightning2', 256),
    ('Lightning7', 128),
    ('ECG200', 32),
    ('Plane', 64),
    ('Car', 256),
    ('Beef', 128),
    ('Coffee', 128),
    ('OliveOil', 256)
]

# We will compare the accuracies & execution times of 1-NN using:
# (i) MINDIST on SAX representations, and
# (ii) euclidean distance on raw values
knn_sax = KNeighborsTimeSeriesClassifier(n_neighbors=1, metric='sax')
knn_eucl = KNeighborsTimeSeriesClassifier(n_neighbors=1, metric='euclidean')

accuracies = {}
times = {}
for dataset, w in datasets:
    X_train, y_train, X_test, y_test = data_loader.load_dataset(dataset)

    ts_scaler = TimeSeriesScalerMeanVariance()
    X_train = ts_scaler.fit_transform(X_train)
    X_test = ts_scaler.fit_transform(X_test)

    # Fit 1-NN using SAX representation & MINDIST
    metric_params = {'n_segments': w, 'alphabet_size_avg': 10}
    knn_sax = clone(knn_sax).set_params(metric_params=metric_params)
    start = time.time()
    knn_sax.fit(X_train, y_train)
    acc_sax = accuracy_score(y_test, knn_sax.predict(X_test))
    time_sax = time.time() - start

    # Fit 1-NN using euclidean distance on raw values
    start = time.time()
    knn_eucl.fit(X_train, y_train)
    acc_euclidean = accuracy_score(y_test, knn_eucl.predict(X_test))
    time_euclidean = time.time() - start

    accuracies[dataset] = (acc_sax, acc_euclidean)
```

(continues on next page)

(continued from previous page)

```
    times[dataset] = (time_sax, time_euclidean)

print_table(accuracies, times)
```

Total running time of the script: (0 minutes 7.235 seconds)

4.6.3 Clustering and Barycenters

DBSCAN

This example illustrates density-based spatial clustering of applications with noise (DBSCAN) for time series. This method, based on finding high density cores, does not require specifying the number of clusters and can identify outliers. The implementation relies on scikit-learn DBSCAN with time series metrics support.

```
import numpy as np

from sklearn.pipeline import Pipeline

import matplotlib.pyplot as plt

from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.clustering import TimeSeriesDBSCAN
from tslearn.datasets import CachedDatasets

X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Keep only timeseries of class 1 and 2 plus 1 outlier from class 3
X_train = np.concatenate((
    X_train[y_train == 1][:10],
    X_train[y_train == 2][:10],
    X_train[y_train == 3][0].reshape(1, -1, 1),
))
y_train = np.concatenate((
    y_train[y_train == 1][:10],
    y_train[y_train == 2][:10],
    np.array([3]),
))
```

Estimator fitting

```
model = Pipeline([
    ('normalize', TimeSeriesScalerMinMax()),
    ('dbscan', TimeSeriesDBSCAN(eps=0.4, min_ts=10))
    ])
model = model.fit(X_train)
labels = model[-1].labels_
core_ts_indices = model[-1].core_ts_indices_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)
```

(continues on next page)

(continued from previous page)

```
print("Estimated number of clusters: %d" % n_clusters_)
print("Estimated number of noise points: %d" % n_noise_)
```

```
Estimated number of clusters: 2
Estimated number of noise points: 1
```

Core and noise

```
fig = plt.figure(figsize=(12, 4),layout="compressed")

core_samples_mask = np.full_like(labels, False, dtype=bool)
core_samples_mask[model[-1].core_ts_indices_] = True

colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, n_clusters_)]

for label in np.unique(labels):
    if label==-1:
        plt.plot(X_train[labels==label].squeeze(), ":k", alpha=1, label="Outlier")
    else:
        plt.plot(X_train[(labels == label) & core_samples_mask].squeeze().T,␣
↪color=colors[label], alpha=0.5,
                 ls='-', label=f"Core TS of cluster {label}")
        plt.plot(X_train[(labels == label) & ~core_samples_mask].squeeze().T,␣
↪color=colors[label], alpha=0.5,
                 ls='--', label=f"Non core TS of cluster {label}")

handles, labels = plt.gca().get_legend_handles_labels()
by_label = dict(zip(labels, handles))
plt.legend(by_label.values(), by_label.keys(), loc='lower right')

fig.suptitle("Clustering results")
plt.show()
```

Total running time of the script: (0 minutes 0.201 seconds)

Soft-DTW weighted barycenters

This example presents the weighted Soft-DTW time series barycenter method.

Soft-DTW[1] is a differentiable loss function for Dynamic Time Warping, allowing for the use of gradient-based algorithms. The barycenter corresponds to the time series that minimizes the sum of the distances between that time series and all the time series from a dataset. It is thus an optimization problem and having a differentiable loss function makes find the solution much easier.

In this example, we consider four time series X_0, X_1, X_2 and X_3 from four different classes in the Trace dataset. We compute the barycenters for different sets of weights and plot them. The closer to a time series the barycenter is, the higher the weight for this time series is.

```
# Author: Romain Tavenard
# License: BSD 3 clause
```

(continues on next page)

[1] M. Cuturi and M. Blondel, "Soft-DTW: a Differentiable Loss Function for Time-Series". International Conference on Machine Learning, 2017.

(continued from previous page)

```
import numpy
import matplotlib.pyplot as plt
import matplotlib.colors

from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.barycenters import softdtw_barycenter
from tslearn.datasets import CachedDatasets


def row_col(position, n_cols=5):
    idx_row = (position - 1) // n_cols
    idx_col = position - n_cols * idx_row - 1
    return idx_row, idx_col


def get_color(weights):
    baselines = numpy.zeros((4, 3))
    weights = numpy.array(weights).reshape(1, 4)
    for i, c in enumerate(["r", "g", "b", "y"]):
        baselines[i] = matplotlib.colors.ColorConverter().to_rgb(c)
    return numpy.dot(weights, baselines).ravel()


numpy.random.seed(0)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
X_out = numpy.empty((4, X_train.shape[1], X_train.shape[2]))

plt.figure()
for i in range(4):
    X_out[i] = X_train[y_train == (i + 1)][0]
X_out = TimeSeriesScalerMinMax().fit_transform(X_out)

for i, pos in enumerate([1, 5, 21, 25]):
    plt.subplot(5, 5, pos)
    w = [0.] * 4
    w[i] = 1.
    plt.plot(X_out[i].ravel(),
             color=matplotlib.colors.rgb2hex(get_color(w)),
             linewidth=2)
    plt.text(X_out[i].shape[0], 0., "$X_%d$" % i,
             horizontalalignment="right",
             verticalalignment="baseline",
             fontsize=24)
    plt.xticks([])
    plt.yticks([])

for pos in range(2, 25):
    if pos in [1, 5, 21, 25]:
        continue
    plt.subplot(5, 5, pos)
    idxr, idxc = row_col(pos, 5)
```

(continues on next page)

(continued from previous page)

```
    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 5.268 seconds)

Barycenters

This example shows three methods to compute barycenters of time series. For an overview over the available methods see the *tslearn.barycenters* module.

tslearn provides three methods for calculating barycenters for a given set of time series:

- *Euclidean barycenter* is simply the arithmetic mean for each individual point in time, minimizing the summed euclidean distance for each of them. As can be seen below, it is very different from the DTW-based methods and may often be inappropriate. However, it is the fastest of the methods shown.
- *DTW Barycenter Averaging (DBA)* is an iteratively refined barycenter, starting out with a (potentially) bad candidate and improving it until convergence criteria are met. The optimization can be accomplished with (a) expectation-maximization[1] and (b) stochastic subgradient descent[2]. Empirically, the latter "is [often] more stable and finds better solutions in shorter time"[2].
- *Soft-DTW barycenter* uses a differentiable loss function to iteratively find a barycenter[3]. The method itself and the parameter $\gamma = 1.0$ is described in more detail in the section on *DTW*. There is also a dedicated *example* available.

```
# Author: Romain Tavenard, Felix Divo
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.barycenters import \
    euclidean_barycenter, \
    dtw_barycenter_averaging, \
    dtw_barycenter_averaging_subgradient, \
    softdtw_barycenter
from tslearn.datasets import CachedDatasets
```

(continues on next page)

[1] F. Petitjean, A. Ketterlin & P. Gancarski. A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition, Elsevier, 2011, Vol. 44, Num. 3, pp. 678-693.

[2] D. Schultz & B. Jain. Nonsmooth Analysis and Subgradient Methods for Averaging in Dynamic Time Warping Spaces. Pattern Recognition, 74, 340-358.

[3] M. Cuturi & M. Blondel. Soft-DTW: a Differentiable Loss Function for Time-Series. ICML 2017.

(continued from previous page)

```
# fetch the example data set
numpy.random.seed(0)
X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")
X = X_train[y_train == 2]
length_of_sequence = X.shape[1]


def plot_helper(barycenter):
    # plot all points of the data set
    for series in X:
        plt.plot(series.ravel(), "k-", alpha=.2)
    # plot the given barycenter of them
    plt.plot(barycenter.ravel(), "r-", linewidth=2)


# plot the four variants with the same number of iterations and a tolerance of
# 1e-3 where applicable
ax1 = plt.subplot(4, 1, 1)
plt.title("Euclidean barycenter")
plot_helper(euclidean_barycenter(X))

plt.subplot(4, 1, 2, sharex=ax1)
plt.title("DBA (vectorized version of Petitjean's EM)")
plot_helper(dtw_barycenter_averaging(X, max_iter=50, tol=1e-3))

plt.subplot(4, 1, 3, sharex=ax1)
plt.title("DBA (subgradient descent approach)")
plot_helper(dtw_barycenter_averaging_subgradient(X, max_iter=50, tol=1e-3))

plt.subplot(4, 1, 4, sharex=ax1)
plt.title("Soft-DTW barycenter ($\\gamma$=1.0)")
plot_helper(softdtw_barycenter(X, gamma=1., max_iter=50, tol=1e-3))

# clip the axes for better readability
ax1.set_xlim([0, length_of_sequence])

# show the plot(s)
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 3.938 seconds)

Kernel k-means

This example uses Global Alignment kernel (GAK,[1]) at the core of a kernel k-means algorithm[2] to perform time series clustering.

Note that, contrary to k-means, a centroid cannot be computed when using kernel k-means. However, one can still report cluster assignments, which is what is provided here: each subfigure represents the set of time series from the training set that were assigned to the considered cluster.

[1] M. Cuturi, "Fast global alignment kernels", ICML 2011.
[2] I. S. Dhillon, Y. Guan, B. Kulis. Kernel k-means, Spectral Clustering and Normalized Cuts. KDD 2004.

```
Init 1
69.989 --> 50.555 --> 44.809 --> 39.228 --> 38.525 --> 38.274 --> 38.274 -->
Init 2
73.500 --> 56.163 --> 46.889 --> 38.525 --> 38.274 --> 38.274 -->
Init 3
72.522 --> 58.699 --> 48.879 --> 39.658 --> 38.525 --> 38.274 --> 38.274 -->
Init 4
72.724 --> 48.397 --> 47.367 --> 45.471 --> 40.415 --> 38.274 --> 38.274 -->
Init 5
67.830 --> 49.865 --> 43.579 --> 40.913 --> 39.228 --> 38.525 --> 38.274 --> 38.274 -->
Init 6
69.032 --> 48.300 --> 38.274 --> 38.274 -->
Init 7
73.764 --> 44.286 --> 40.062 --> 39.228 --> 38.525 --> 38.274 --> 38.274 -->
Init 8
70.647 --> 43.591 --> 38.789 --> 38.274 --> 38.274 -->
Init 9
69.114 --> 50.962 --> 46.566 --> 41.645 --> 38.274 --> 38.274 -->
Init 10
72.032 --> 50.975 --> 40.080 --> 38.525 --> 38.274 --> 38.274 -->
Init 11
68.385 --> 39.940 --> 38.525 --> 38.274 --> 38.274 -->
Init 12
71.264 --> 54.645 --> 53.469 --> 50.041 --> 47.748 --> 47.367 --> 45.471 --> 40.415 -->␣
↪38.274 --> 38.274 -->
Init 13
68.948 --> 56.189 --> 55.228 --> 51.461 --> 47.355 --> 40.913 --> 39.228 --> 38.525 -->␣
↪38.274 --> 38.274 -->
Init 14
70.825 --> 39.433 --> 38.274 --> 38.274 -->
Init 15
69.938 --> 53.940 --> 53.139 --> 51.893 --> 51.337 --> 51.337 -->
Init 16
73.047 --> 57.085 --> 55.936 --> 54.749 --> 52.264 --> 45.080 --> 39.497 --> 38.525 -->␣
↪38.274 --> 38.274 -->
Init 17
69.231 --> 50.839 --> 41.550 --> 38.274 --> 38.274 -->
Init 18
68.292 --> 58.990 --> 56.241 --> 53.241 --> 45.728 --> 40.138 --> 38.274 --> 38.274 -->
Init 19
69.969 --> 40.015 --> 38.274 --> 38.274 -->
Init 20
69.511 --> 51.434 --> 48.272 --> 47.017 --> 45.178 --> 40.415 --> 38.274 --> 38.274 -->
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt

from tslearn.clustering import KernelKMeans
from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

seed = 0
numpy.random.seed(seed)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
# Keep first 3 classes
X_train = X_train[y_train < 4]
numpy.random.shuffle(X_train)
# Keep only 50 time series
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train[:50])
sz = X_train.shape[1]

gak_km = KernelKMeans(n_clusters=3,
                      kernel="gak",
                      kernel_params={"sigma": "auto"},
                      n_init=20,
                      verbose=True,
                      random_state=seed)
y_pred = gak_km.fit_predict(X_train)

plt.figure()
for yi in range(3):
    plt.subplot(3, 1, 1 + yi)
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-", alpha=.2)
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.title("Cluster %d" % (yi + 1))

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 2.312 seconds)

k-means

This example uses k-means clustering for time series. Three variants of the algorithm are available: standard Euclidean k-means, DBA-k-means (for DTW Barycenter Averaging[1]) and Soft-DTW k-means[2].

In the figure below, each row corresponds to the result of a different clustering. In a row, each sub-figure corresponds to a cluster. It represents the set of time series from the training set that were assigned to the considered cluster (in black) as well as the barycenter of the cluster (in red).

[1] F. Petitjean, A. Ketterlin & P. Gancarski. A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition, Elsevier, 2011, Vol. 44, Num. 3, pp. 678-693

[2] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series", ICML 2017.

A note on pre-processing

In this example, time series are preprocessed using *TimeSeriesScalerMeanVariance*. This scaler is such that each output time series has zero mean and unit variance. The assumption here is that the range of a given time series is uninformative and one only wants to compare shapes in an amplitude-invariant manner (when time series are multivariate, this also rescales all modalities such that there will not be a single modality responsible for a large part of the variance). This means that one cannot scale barycenters back to data range because each time series is scaled independently and there is hence no such thing as an overall data range.

```
Euclidean k-means
15.795 --> 7.716 --> 7.716 -->
DBA k-means
Init 1
0.637 --> 0.458 --> 0.458 -->
Init 2
0.826 --> 0.525 --> 0.477 --> 0.472 --> 0.472 -->
Soft-DTW k-means
0.472 --> 0.144 --> 0.142 --> 0.143 --> 0.142 --> 0.143 --> 0.142 --> 0.143 --> 0.142 -->
↪ 0.142 --> 0.142 --> 0.142 --> 0.142 --> 0.142 --> 0.142 --> 0.142 --> 0.142 --> 0.142␣
↪--> 0.142 --> 0.142 --> 0.142 -->
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt

from tslearn.clustering import TimeSeriesKMeans
from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance, \
    TimeSeriesResampler

seed = 0
numpy.random.seed(seed)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
X_train = X_train[y_train < 4]  # Keep first 3 classes
numpy.random.shuffle(X_train)
# Keep only 50 time series
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train[:50])
# 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)
```

(continues on next page)

(continued from previous page)

```
plt.figure()
for yi in range(3):
    plt.subplot(3, 3, yi + 1)
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-", alpha=.2)
    plt.plot(km.cluster_centers_[yi].ravel(), "r-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.text(0.55, 0.85,'Cluster %d' % (yi + 1),
             transform=plt.gca().transAxes)
    if yi == 1:
        plt.title("Euclidean $k$-means")

# DBA-k-means
print("DBA k-means")
dba_km = TimeSeriesKMeans(n_clusters=3,
                          n_init=2,
                          metric="dtw",
                          verbose=True,
                          max_iter_barycenter=10,
                          random_state=seed)
y_pred = dba_km.fit_predict(X_train)

for yi in range(3):
    plt.subplot(3, 3, 4 + yi)
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-", alpha=.2)
    plt.plot(dba_km.cluster_centers_[yi].ravel(), "r-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.text(0.55, 0.85,'Cluster %d' % (yi + 1),
             transform=plt.gca().transAxes)
    if yi == 1:
        plt.title("DBA $k$-means")

# Soft-DTW-k-means
print("Soft-DTW k-means")
sdtw_km = TimeSeriesKMeans(n_clusters=3,
                           metric="softdtw",
                           metric_params={"gamma": .01},
                           verbose=True,
                           random_state=seed)
y_pred = sdtw_km.fit_predict(X_train)

for yi in range(3):
    plt.subplot(3, 3, 7 + yi)
    for xx in X_train[y_pred == yi]:
        plt.plot(xx.ravel(), "k-", alpha=.2)
    plt.plot(sdtw_km.cluster_centers_[yi].ravel(), "r-")
    plt.xlim(0, sz)
    plt.ylim(-4, 4)
    plt.text(0.55, 0.85,'Cluster %d' % (yi + 1),
```

(continues on next page)

(continued from previous page)

```
            transform=plt.gca().transAxes)
    if yi == 1:
        plt.title("Soft-DTW $k$-means")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 10.035 seconds)

KShape

This example uses the KShape clustering method[1] that is based on cross-correlation to cluster time series.

```
0.008 --> 0.006 --> 0.005 --> 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 --> 0.003 --> 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 --> 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 --> 0.002 -->
↪ 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.001␣
↪--> 0.001 --> 0.001 --> 0.001 --> 0.001 --> 0.001 --> 0.001 --> 0.001 -->
```

```
# 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

numpy.random.seed(0)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")
# Keep first 3 classes and 50 first time series
X_train = X_train[y_train < 4]
X_train = X_train[:50]
numpy.random.shuffle(X_train)
# For this method to operate properly, prior scaling is required
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train)
sz = X_train.shape[1]

# kShape clustering
init=numpy.array([X_train[44], X_train[47], X_train[0]])
ks = KShape(n_clusters=3, verbose=True, init=init)
```

(continues on next page)

[1] J. Paparrizos & L. Gravano. k-Shape: Efficient and Accurate Clustering of Time Series. SIGMOD 2015. pp. 1855-1870.

(continued from previous page)

```
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 24.775 seconds)

4.6.4 Classification

Early Classification

This example presents the concept of early classification.

Early classifiers are implemented in the *tslearn.early_classification* module and in this example we use the method from[1].

References

```
# Author: Romain Tavenard
# License: BSD 3 clause
# sphinx_gallery_thumbnail_number = 2

from contextlib import suppress

import numpy

import matplotlib.animation as animation
import matplotlib.gridspec as gridpsec
import matplotlib.pyplot as plt

from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn.early_classification import NonMyopicEarlyClassifier
from tslearn.datasets import UCR_UEA_datasets

def plot_partial(time_series, t, y_true=0, y_pred=0, color="k"):
    plt.plot(time_series[:t+1].ravel(), color=color, linewidth=1.5)
    plt.plot(numpy.arange(t+1, time_series.shape[0]),
             time_series[t+1:].ravel(),
             linestyle="dashed", color=color, linewidth=1.5)
    plt.axvline(x=t, color=color, linewidth=1.5)
```

(continues on next page)

[1] A. Dachraoui, A. Bondu & A. Cornuejols. Early classification of time series as a non myopic sequential decision making problem. ECML/PKDD 2015

(continued from previous page)

```
    plt.text(x=t - 20, y=time_series.max() - .25, s="Prediction time")
    plt.title(
        "Sample of class {} predicted as class {}".format(y_true, y_pred)
    )
    plt.xlim(0, time_series.shape[0] - 1)
```

Data loading and visualization

```
numpy.random.seed(0)
loader = UCR_UEA_datasets()
X_train, y_train, X_test, y_test = loader.load_dataset("ECG200")

# Scale time series
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train)
X_test = TimeSeriesScalerMeanVariance().fit_transform(X_test)

size = X_train.shape[1]
n_classes = len(set(y_train))

plt.figure(layout="constrained")
for i, cl in enumerate(set(y_train)):
    ax = plt.subplot(n_classes, 1, i + 1)
    ax.set_title(f"Class {cl}")
    for ts in X_train[y_train == cl]:
        plt.plot(ts.ravel(), color="orange" if cl > 0 else "blue", alpha=.3)
    plt.xlim(0, size - 1)
plt.suptitle("Training time series")
plt.show()
```

Model fitting

As observed in the following figure, the optimal classification time as estimated by *NonMyopicEarlyClassifier* is data-dependent.

```
early_clf = NonMyopicEarlyClassifier(n_clusters=3,
                                     cost_time_parameter=1e-3,
                                     lamb=1e2,
                                     random_state=0)
early_clf.fit(X_train, y_train)

preds, times = early_clf.predict_class_and_earliness(X_test)

plt.figure()
plt.subplot(2, 1, 1)
ts_idx = 0
t = times[ts_idx]
plot_partial(X_test[ts_idx], t, y_test[ts_idx], preds[ts_idx], color="orange")


plt.subplot(2, 1, 2)
```

(continues on next page)

(continued from previous page)

```
ts_idx = 9
t = times[ts_idx]
plot_partial(X_test[ts_idx], t, y_test[ts_idx], preds[ts_idx], color="blue")
plt.tight_layout()
plt.show()
```

Streaming inputs

Let's focus on analyzing early classification of a time series acquired sequentially over time.

For each incoming timestamp t, the following figure displays data-dependant computations of:

- the clustering probabilities $P(C_k|\mathbf{x}_{\to t})$
- the expected cost for all future times $t + \tau$ with $\tau \geq 0$:

$$f_\tau(\mathbf{x}_{\to t}, y) = \sum_k \left[P(C_k|\mathbf{x}_{\to t}) \sum_i \left(P(y=i|C_k) \left(\sum_{j \neq i} P_{t+\tau}(\hat{y}=j|y=i, C_k) \right) \right) \right] + \alpha t$$

as described in *our User Guide section dedicated to early classification*.

The estimated optimal τ is derived at each timestamp from minimizing the expected costs.

In this example, the *NonMyopicEarlyClassifier* recommends a classification decision as early as $t = 12$.

```
ts_index = 1
sz = X_test.shape[1]

fig = plt.figure(layout="constrained", figsize=(13, 4))
fig.suptitle(r"Optimal prediction time $\tau$ evolution")

gs = gridpsec.GridSpec(2, 3, figure=fig, width_ratios=[0.15, 0.70, 0.15])
ax1 = fig.add_subplot(gs[:, 0], title='Cluster probas')
ax2 = fig.add_subplot(
    gs[0, 1],
    xlim=[0, sz],
    ylim=[numpy.min(X_test[ts_index]),
          numpy.max(X_test[ts_index]) * 1.1],
    title='Streamed TS'
)
ax3 = fig.add_subplot(gs[1, 1], xlim=[0, sz], ylim=[0, 1], title='Expected cost')
ax4 = fig.add_subplot(gs[:, 2], title='Predicted probas')

bar1 = ax1.barh(
    ["cluster 1", "cluster 2", "cluster 3"],
    [1.1, 0, 0],
)
line1 = ax2.plot([numpy.nan], marker='.')[0]
line2 = ax3.plot(numpy.full((sz,), numpy.nan), linestyle="--",  marker='.')[0]
bar2 = ax4.bar(
    ["class -1", "class 1"],
    [1.1, 0],
)
```

(continues on next page)

(continued from previous page)

```
def update(frame):
    incoming_ts_ =  X_test[ts_index, :frame+1]
    cluster_probas = early_clf.get_cluster_probas(incoming_ts_)
    expected_costs = early_clf._expected_costs(incoming_ts_).reshape(-1)
    probas, delays = early_clf.early_predict_proba(
        numpy.expand_dims(incoming_ts_, axis=0)
    )
    proba, delay = probas[0], delays[0]

    for i, elem in enumerate(bar1):
        elem.set_width(cluster_probas[i])

    line1.set_xdata(numpy.arange(incoming_ts_.shape[0]))
    line1.set_ydata(incoming_ts_)

    for i, elem in enumerate(bar2):
        elem.set_height(proba[i])

    with suppress(IndexError):
        ax2.lines[1].remove()
        ax3.lines[1].remove()
        ax2.texts[0].remove()
    ax2.axvline(x=frame + delay, color="k", linewidth=1.5)
    ax3.axvline(x=frame + delay, color="k", linewidth=1.5)
    ax2.text(x=frame + delay, y= numpy.max(X_test[ts_index])/2, s=r"$\tau$")
    line2.set_xdata(numpy.arange(expected_costs.shape[0]) + frame)
    line2.set_ydata(expected_costs)
    return bar1, line1, line2, bar2

ani = animation.FuncAnimation(fig=fig, func=update, frames=sz, interval=100)
plt.show()
```

Earliness-Accuracy trade-off

The trade-off between earliness and accuracy is controlled via `cost_time_parameter`.

```
plt.figure()
hatches = ["///", "\\\\\\", "*"]
for i, cost_t in enumerate([1e-4, 1e-3, 1e-2]):
    early_clf.set_params(cost_time_parameter=cost_t)
    early_clf.fit(X_train, y_train)
    preds, times = early_clf.predict_class_and_earliness(X_test)
    plt.hist(times,
             alpha=.5, hatch=hatches[i],
             density=True,
             label="$\\alpha={}$".format(cost_t),
             bins=numpy.arange(0, size, 5))
plt.legend(loc="upper right")
plt.xlim(0, size - 1)
plt.xlabel("Prediction times")
plt.title("Impact of cost_time_parameter ($\\alpha$)")
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

Total running time of the script: (1 minutes 36.311 seconds)

Learning Shapelets: decision boundaries in 2D distance space

This example illustrates the use of the "Learning Shapelets" method, presented in[1], in order to learn a collection of shapelets that linearly separates the timeseries. In this example, we will extract two shapelets which are then used to transform our input time series in a two-dimensional space, which is called the shapelet-transform space in the related literature. Moreover, we plot the decision boundaries of our classifier for each of the different classes.

References

```
# Author: Gilles Vandewiele
# License: BSD 3 clause

import os

# Should be set before importing keras
os.environ["KERAS_BACKEND"] = "torch"

from keras.optimizers import Adam
import numpy
from matplotlib import cm
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.shapelets import LearningShapelets

# Load the Trace dataset
X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Normalize the time series
X_train = TimeSeriesScalerMinMax().fit_transform(X_train)

# Get statistics of the dataset
n_ts, ts_sz = X_train.shape[:2]
n_classes = len(set(y_train))

# We will extract 2 shapelets and align them with the time series
shapelet_sizes = {20: 2}

# Define the model and fit it using the training data
shp_clf = LearningShapelets(n_shapelets_per_size=shapelet_sizes,
                            weight_regularizer=0.0001,
                            optimizer=Adam(0.01),
                            max_iter=300,
                            verbose=0,
```

(continues on next page)

[1] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

(continued from previous page)

```
                            scale=False,
                            random_state=0)
shp_clf.fit(X_train, y_train)

# We will plot our distances in a 2D space
distances = shp_clf.transform(X_train).reshape((-1, 2))
weights, biases = shp_clf.get_weights('classification')

# Create a grid for our two shapelets on the left and distances on the right
viridis = cm.get_cmap('viridis', 4)
fig = plt.figure(constrained_layout=True)
gs = fig.add_gridspec(3, 9)
fig_ax1 = fig.add_subplot(gs[0, :2])
fig_ax2 = fig.add_subplot(gs[0, 2:4])
fig_ax3a = fig.add_subplot(gs[1, :2])
fig_ax3b = fig.add_subplot(gs[1, 2:4])
fig_ax3c = fig.add_subplot(gs[2, :2])
fig_ax3d = fig.add_subplot(gs[2, 2:4])
fig_ax4 = fig.add_subplot(gs[:, 4:])

# Plot our two shapelets on the left side
fig_ax1.plot(shp_clf.shapelets_[0])
fig_ax1.set_title('Shapelet $\\mathbf{s}_1$')

fig_ax2.plot(shp_clf.shapelets_[1])
fig_ax2.set_title('Shapelet $\\mathbf{s}_2$')

# Create the time series of each class
for i, subfig in enumerate([fig_ax3a, fig_ax3b, fig_ax3c, fig_ax3d]):
    for k, ts in enumerate(X_train[y_train == i + 1]):
        subfig.plot(ts.flatten(), c=viridis(i / 3), alpha=0.25)
        subfig.set_title('Class {}'.format(i + 1))
fig.text(x=.15, y=.02, s='Input time series', fontsize=12)

# Create a scatter plot of the 2D distances for the time series of each class.
for i, y in enumerate(numpy.unique(y_train)):
    fig_ax4.scatter(distances[y_train == y][:, 0],
                    distances[y_train == y][:, 1],
                    c=[viridis(i / 3)] * numpy.sum(y_train == y),
                    edgecolors='k',
                    label='Class {}'.format(y))

# Create a meshgrid of the decision boundaries
xmin = numpy.min(distances[:, 0]) - 0.1
xmax = numpy.max(distances[:, 0]) + 0.1
ymin = numpy.min(distances[:, 1]) - 0.1
ymax = numpy.max(distances[:, 1]) + 0.1
xx, yy = numpy.meshgrid(numpy.arange(xmin, xmax, (xmax - xmin)/200),
                        numpy.arange(ymin, ymax, (ymax - ymin)/200))
Z = []
for x, y in numpy.c_[xx.ravel(), yy.ravel()]:
    Z.append(numpy.argmax([biases[i] + weights[0][i]*x + weights[1][i]*y
```

(continues on next page)

(continued from previous page)

```
                            for i in range(4)]))
Z = numpy.array(Z).reshape(xx.shape)
cs = fig_ax4.contourf(xx, yy, Z / 3, cmap=viridis, alpha=0.25)

fig_ax4.legend()
fig_ax4.set_xlabel('$d(\\mathbf{x}, \\mathbf{s}_1)$')
fig_ax4.set_ylabel('$d(\\mathbf{x}, \\mathbf{s}_2)$')
fig_ax4.set_xlim((xmin, xmax))
fig_ax4.set_ylim((ymin, ymax))
fig_ax4.set_title('Distance transformed time series')
plt.show()
```

Total running time of the script: (0 minutes 22.606 seconds)

Aligning discovered shapelets with timeseries

This example illustrates the use of the "Learning Shapelets" method, presented in[1] , in order to learn a collection of shapelets that linearly separates the timeseries. In this example, we will extract a single shapelet in order to distinguish between two classes of the "Trace" dataset. Afterwards, we show how our time series can be transformed to distances by aligning the shapelets along each of the time series. This alignment is performed by shifting the smaller shapelet across the longer time series and taking the minimal pointwise distance.

References

```
# Author: Romain Tavenard
# License: BSD 3 clause

import os

# Should be set before importing keras through
os.environ["KERAS_BACKEND"] = "torch"

from keras.optimizers import Adam
import numpy
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.shapelets import LearningShapelets

# Load the Trace dataset
X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Filter out classes 2 and 4
mask = numpy.isin(y_train, [1, 3])
X_train = X_train[mask]
y_train = y_train[mask]

# Normalize the time series
X_train = TimeSeriesScalerMinMax().fit_transform(X_train)
```

(continues on next page)

[1] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

(continued from previous page)

```
# Get statistics of the dataset
n_ts, ts_sz = X_train.shape[:2]
n_classes = len(set(y_train))

# We will extract 1 shapelet and align it with a time series
shapelet_sizes = {20: 1}

# Define the model and fit it using the training data
shp_clf = LearningShapelets(n_shapelets_per_size=shapelet_sizes,
                            weight_regularizer=0.001,
                            optimizer=Adam(0.01),
                            max_iter=250,
                            verbose=0,
                            scale=False,
                            random_state=42)
shp_clf.fit(X_train, y_train)

# Get the number of extracted shapelets, the (minimal) distances from
# each of the timeseries to each of the shapelets, and the corresponding
# locations (index) where the minimal distance was found
n_shapelets = sum(shapelet_sizes.values())
distances = shp_clf.transform(X_train)
predicted_locations = shp_clf.locate(X_train)

f, ax = plt.subplots(2, 1, sharex=True)

# Plot the shapelet and align it on the best matched time series. The optimizer
# will often enlarge the shapelet to create a larger gap between the distances
# of both classes. We therefore normalize the shapelet again before plotting.
test_ts_id = numpy.argmin(numpy.sum(distances, axis=1))
shap = shp_clf.shapelets_[0]
shap = TimeSeriesScalerMinMax().fit_transform(shap.reshape(1, -1, 1)).flatten()
pos = predicted_locations[test_ts_id, 0]
ax[0].plot(X_train[test_ts_id].ravel())
ax[0].plot(numpy.arange(pos, pos + len(shap)), shap, linewidth=2)
ax[0].axvline(pos, color='k', linestyle='--', alpha=0.25)
ax[0].set_title("The aligned extracted shapelet")

# We calculate the distances from the shapelet to the timeseries ourselves.
distances = []
time_series = X_train[test_ts_id].ravel()
for i in range(len(time_series) - len(shap)):
    distances.append(numpy.linalg.norm(time_series[i:i+len(shap)] - shap))
ax[1].plot(distances)
ax[1].axvline(numpy.argmin(distances), color='k', linestyle='--', alpha=0.25)
ax[1].set_title('The distances between the time series and the shapelet')

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 20.057 seconds)

Learning Shapelets

This example illustrates how the "Learning Shapelets" method, presented in[1], can quicklyfind a set of shapelets that results in excellent predictive performance when used for a shapelet transform.

References

-
-


```
Correct classification rate: 1.0
```

```
# Author: Romain Tavenard
# License: BSD 3 clause

import os

# Should be set before importing keras
os.environ["KERAS_BACKEND"] = "torch"

from keras.optimizers import Adam
import numpy
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMinMax
from tslearn.shapelets import LearningShapelets, \
    grabocka_params_to_shapelet_size_dict
from tslearn.utils import ts_size

# Load the Trace dataset
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")

# Normalize each of the timeseries in the Trace dataset
X_train = TimeSeriesScalerMinMax().fit_transform(X_train)
X_test = TimeSeriesScalerMinMax().fit_transform(X_test)

# Get statistics of the dataset
n_ts, ts_sz = X_train.shape[:2]
n_classes = len(set(y_train))

# Set the number of shapelets per size as done in the original paper
shapelet_sizes = grabocka_params_to_shapelet_size_dict(n_ts=n_ts,
                                                       ts_sz=ts_sz,
                                                       n_classes=n_classes,
                                                       l=0.1,
```

(continues on next page)

[1] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

(continued from previous page)

```
                                                    r=1)

# Define the model using parameters provided by the authors (except that we
# use fewer iterations here)
shp_clf = LearningShapelets(n_shapelets_per_size=shapelet_sizes,
                            optimizer=Adam(0.01),
                            batch_size=16,
                            weight_regularizer=0.01,
                            max_iter=200,
                            random_state=0,
                            verbose=0)
shp_clf.fit(X_train, y_train)

# Make predictions and calculate accuracy score
pred_labels = shp_clf.predict(X_test)
print("Correct classification rate:", accuracy_score(y_test, pred_labels))

# Plot the different discovered shapelets
plt.figure()
for i, sz in enumerate(shapelet_sizes.keys()):
    plt.subplot(len(shapelet_sizes), 1, i + 1)
    plt.title("%d shapelets of size %d" % (shapelet_sizes[sz], sz))
    for shp in shp_clf.shapelets_:
        if ts_size(shp) == sz:
            plt.plot(shp.ravel())
    plt.xlim([0, max(shapelet_sizes.keys()) - 1])

plt.tight_layout()
plt.show()

# The loss history is accessible via the `model_` that is a keras model
plt.figure()
plt.plot(numpy.arange(1, shp_clf.n_iter_ + 1), shp_clf.history_["loss"])
plt.title("Evolution of cross-entropy loss during training")
plt.xlabel("Epochs")
plt.show()
```

Total running time of the script: (0 minutes 31.613 seconds)

SVM and GAK

This example illustrates the use of the global alignment kernel (GAK) for support vector classification.

This metric is defined in the *tslearn.metrics* module and explained in details in[1].

In this example, a *TimeSeriesSVC* model that uses GAK as kernel is fit and the support vectors for each class are reported.

```
Correct classification rate: 1.0
```

[1] M. Cuturi, "Fast global alignment kernels", ICML 2011.

```
# 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_
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: (0 minutes 22.453 seconds)

4.6.5 Automatic differentiation

Soft-DTW loss for PyTorch neural network

The aim here is to use the Soft Dynamic Time Warping metric as a loss function of a PyTorch Neural Network for time series forecasting.

The *torch*-compatible implementation of the soft-DTW loss function is available from the `tslearn.metrics` module.

```
# Authors: Yann Cabanes, Romain Tavenard
# License: BSD 3 clause
# sphinx_gallery_thumbnail_number = 2

"""Import the modules"""

import numpy as np
import matplotlib.pyplot as plt
from tslearn.datasets import CachedDatasets
from tslearn.metrics import SoftDTWLossPyTorch
import torch
```

(continues on next page)

(continued from previous page)

```
from torch import nn
```

Load the dataset

Using the CachedDatasets utility from tslearn, we load the "Trace" time series dataset. The dimensions of the arrays storing the time series training and testing datasets are (100, 275, 1). We create a new dataset X_subset made of 50 random time series from classes indexed 1 to 3 (y_train < 4) in the training set: X_subset is of shape (50, 275, 1).

```
data_loader = CachedDatasets()
X_train, y_train, X_test, y_test = data_loader.load_dataset("Trace")

X_subset = X_train[y_train < 4]
np.random.shuffle(X_subset)
X_subset = X_subset[:50]
```

Multi-step ahead forecasting

In this section, our goal is to implement a single-hidden-layer perceptron for time series forecasting. Our network will be trained to minimize the soft-DTW metric. We will rely on a *torch*-compatible implementation of the soft-DTW loss function. The code below is an implementation of a generic Multi-Layer-Perceptron class in torch, and we will rely on it for the implementation of a forecasting MLP with softDTW loss.

```
# Note that Soft-DTW can take negative values due to the regularization parameter gamma.
# The normalized soft-DTW (also coined soft-DTW divergence) between the time series x␣
↪and y is defined as:
# Soft-DTW(x, y) - (Soft-DTW(x, x) + Soft-DTW(y, y)) / 2
# The normalized Soft-DTW is always positive.
# However, the computation time of the normalized soft-DTW equals three times the␣
↪computation time of the Soft-DTW.

class MultiLayerPerceptron(torch.nn.Module):
    def __init__(self, layers, loss=None):
        # At init, we define our layers
        super(MultiLayerPerceptron, self).__init__()
        self.layers = layers
        if loss is None:
            self.loss = torch.nn.MSELoss(reduction="none")
        else:
            self.loss = loss
        self.optimizer = torch.optim.SGD(self.parameters(), lr=0.001)

    def forward(self, X):
        # The forward method informs about the forward pass: how one computes outputs of␣
↪the network
        # from the input and the parameters of the layers registered at init
        if not isinstance(X, torch.Tensor):
            X = torch.Tensor(X)
        batch_size = X.size(0)
        X_reshaped = torch.reshape(X, (batch_size, -1))  # Manipulations to deal with␣
↪time series format
        output = self.layers(X_reshaped)
```

(continues on next page)

(continued from previous page)

```
        return torch.reshape(output, (batch_size, -1, 1))  # Manipulations to deal with␣
↪time series format

    def fit(self, X, y, max_epochs=10):
        # The fit method performs the actual optimization
        X_torch = torch.Tensor(X)
        y_torch = torch.Tensor(y)

        for e in range(max_epochs):
            self.optimizer.zero_grad()
            # Forward pass
            y_pred = self.forward(X_torch)
            # Compute Loss
            loss = self.loss(y_pred, y_torch).mean()
            # Backward pass
            loss.backward()
            self.optimizer.step()
```

Using MSE as a loss function

We define an MLP class that would allow training a single-hidden-layer model using mean squared error (MSE) as a loss function to be optimized. We train the network for 1000 epochs on a forecasting task that would consist, given the first 150 elements of a time series, in predicting the next 125 ones.

```
model = MultiLayerPerceptron(
    layers=nn.Sequential(
        nn.Linear(in_features=150, out_features=256),
        nn.ReLU(),
        nn.Linear(in_features=256, out_features=125)
    )
)

# Here one needs to define what X and y are, obviously
model.fit(X_subset[:, :150], X_subset[:, 150:], max_epochs=1000)

ts_index = 50
y_pred = model(X_test[:, :150, 0]).detach().numpy()

plt.figure(1, layout="constrained")
plt.title('Multi-step ahead forecasting using MSE')
plt.plot(X_test[ts_index].ravel())
plt.plot(np.arange(150, 275), y_pred[ts_index], 'r-')
plt.show()
```

Using Soft-DTW as a loss function

We take inspiration from the code above to define an MLP class that would allow training a single-hidden-layer model using soft-DTW as a criterion to be optimized. We train the network for 100 epochs on a forecasting task that would consist, given the first 150 elements of a time series, in predicting the next 125 ones.

```
model = MultiLayerPerceptron(
    layers=nn.Sequential(
        nn.Linear(in_features=150, out_features=256),
        nn.ReLU(),
        nn.Linear(in_features=256, out_features=125)
    ),
    loss=SoftDTWLossPyTorch(gamma=0.1)
)

model.fit(X_subset[:, :150], X_subset[:, 150:], max_epochs=100)

y_pred = model(X_test[:, :150, 0]).detach().numpy()

plt.figure(2, layout="constrained")
plt.title('Multi-step ahead forecasting using Soft-DTW loss')
plt.plot(X_test[ts_index].ravel())
plt.plot(np.arange(150, 275), y_pred[ts_index], 'r-')
plt.show()
```

Total running time of the script: (0 minutes 6.500 seconds)

4.6.6 Miscellaneous

Distance and Matrix Profiles

This example illustrates how the matrix profile is calculated. For each segment of a timeseries with a specified length, the distances between each subsequence and that segment are calculated. The smallest distance is returned, except for trivial match on the location where the segment is extracted from which is equal to zero.

```
# Author: Gilles Vandewiele
# License: BSD 3 clause

import numpy
import matplotlib.patches as patches
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
import matplotlib.pyplot as plt

from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn.matrix_profile import MatrixProfile

import warnings
warnings.filterwarnings('ignore')

# Set a seed to ensure determinism
numpy.random.seed(42)

# Load the Trace dataset
X_train, y_train, _, _ = CachedDatasets().load_dataset("Trace")

# Normalize the time series
scaler = TimeSeriesScalerMeanVariance()
```

(continues on next page)

(continued from previous page)

```
X_train = scaler.fit_transform(X_train)

# Take the first time series
ts = X_train[0, :, :]

# We will take the spike as a segment
subseq_len = 20
start = 45
segment = ts[start:start + subseq_len]

# Create our matrix profile
matrix_profiler = MatrixProfile(subsequence_length=subseq_len, scale=True)
mp = matrix_profiler.fit_transform([ts]).flatten()

# Create a grid for our plots
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)

# Plot our timeseries
ax1.plot(ts, c='b', label='time series')
ax1.add_patch(patches.Rectangle((start, numpy.min(ts) - 0.1), subseq_len,
                                numpy.max(ts) - numpy.min(ts) + 0.2,
                                facecolor='b', alpha=0.25,
                                label='segment'))
ax1.axvline(start, c='b', linestyle='--', lw=2, alpha=0.5,
            label='segment start')
ax1.legend(loc='lower right', ncol=4, fontsize=8,
           handletextpad=0.1, columnspacing=0.5)
ax1.set_title('The time series')

# Inset plot with our segment
fig_ax_in = ax1.inset_axes([0.5, 0.55, 0.2, 0.4])
fig_ax_in.plot(scaler.fit_transform(segment.reshape(1, -1, 1))[0], c='b')
ax1.indicate_inset(inset_ax=fig_ax_in, transform=ax1.transData,
                   bounds=[start, numpy.min(ts) - 0.1, subseq_len,
                           numpy.max(ts) - numpy.min(ts) + 0.2],
                   linestyle='--', alpha=0.75)
fig_ax_in.tick_params(labelleft=False, labelbottom=False)
fig_ax_in.xaxis.set_visible(False)
fig_ax_in.yaxis.set_visible(False)

# Calculate a distance profile, which represents the distance from each
# subsequence of the time series and the segment
distances = []
for i in range(len(ts) - subseq_len):
    scaled_ts = scaler.fit_transform(ts[i:i+subseq_len].reshape(1, -1, 1))
    scaled_segment = scaler.fit_transform(segment.reshape(1, -1, 1))
    distances.append(numpy.linalg.norm(scaled_ts - scaled_segment))

# Mask out the distances in the trivial match zone, get the nearest
# neighbor and put the old distances back in place so we can plot them.
distances = numpy.array(distances)
mask = list(range(start - subseq_len // 4, start + subseq_len // 4))
```

(continues on next page)

(continued from previous page)

```
old_distances = distances[mask]
distances[mask] = numpy.inf
nearest_neighbor = numpy.argmin(distances)
dist_nn = distances[nearest_neighbor]
distances[mask] = old_distances

# Plot our distance profile
ax2.plot(distances, c='b')
ax2.set_title('Segment distance profile')
dist_diff = numpy.max(distances) - numpy.min(distances)
ax2.add_patch(patches.Rectangle((start - subseq_len // 4,
                                 numpy.min(distances) - 0.1),
                                subseq_len // 2,
                                dist_diff + 0.2,
                                facecolor='r', alpha=0.5,
                                label='exclusion zone'))
ax2.scatter(nearest_neighbor, dist_nn, c='r', marker='x', s=50,
            label='neighbor dist = {}'.format(numpy.around(dist_nn, 3)))
ax2.axvline(start, c='b', linestyle='--', lw=2, alpha=0.5,
            label='segment start')
ax2.legend(loc='lower right', fontsize=8, ncol=3,
           handletextpad=0.1, columnspacing=0.5)

# Plot our matrix profile
ax3.plot(mp, c='b')
ax3.set_title('Matrix profile')
ax3.scatter(start, mp[start],
            c='r', marker='x', s=75,
            label='MP segment = {}'.format(numpy.around(mp[start], 3)))
ax3.axvline(start, c='b', linestyle='--', lw=2, alpha=0.5,
            label='segment start')
ax3.legend(loc='lower right', fontsize=8,
           handletextpad=0.1, columnspacing=0.25)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.711 seconds)

Matrix Profile

This example presents a toy example of using Matrix Profile[1] for anomaly detection.

Matrix Profile transforms a time series into a sequence of 1-Nearest-Neighbor distances between its subseries.

```
# Author: Romain Tavenard
# License: BSD 3 clause

import numpy
import matplotlib.pyplot as plt
```

(continues on next page)

[1] C. M. Yeh, Y. Zhu, L. Ulanova, N.Begum et al. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. ICDM 2016.

(continued from previous page)

```
import matplotlib.transforms as mtransforms

from tslearn.matrix_profile import MatrixProfile

s_x = numpy.array(
    [-0.790, -0.765, -0.734, -0.700, -0.668, -0.639, -0.612, -0.587, -0.564,
     -0.544, -0.529, -0.518, -0.509, -0.502, -0.494, -0.488, -0.482, -0.475,
     -0.472, -0.470, -0.465, -0.464, -0.461, -0.458, -0.459, -0.460, -0.459,
     -0.458, -0.448, -0.431, -0.408, -0.375, -0.333, -0.277, -0.196, -0.090,
     0.047, 0.220, 0.426, 0.671, 0.962, 1.300, 1.683, 2.096, 2.510, 2.895,
     3.219, 3.463, 3.621, 3.700, 3.713, 3.677, 3.606, 3.510, 3.400, 3.280,
     3.158, 3.038, 2.919, 2.801, 2.676, 2.538, 2.382, 2.206, 2.016, 1.821,
     1.627, 1.439, 1.260, 1.085, 0.917, 0.758, 0.608, 0.476, 0.361, 0.259,
     0.173, 0.096, 0.027, -0.032, -0.087, -0.137, -0.179, -0.221, -0.260,
     -0.293, -0.328, -0.359, -0.385, -0.413, -0.437, -0.458, -0.480, -0.498,
     -0.512, -0.526, -0.536, -0.544, -0.552, -0.556, -0.561, -0.565, -0.568,
     -0.570, -0.570, -0.566, -0.560, -0.549, -0.532, -0.510, -0.480, -0.443,
     -0.402, -0.357, -0.308, -0.256, -0.200, -0.139, -0.073, -0.003, 0.066,
     0.131, 0.186, 0.229, 0.259, 0.276, 0.280, 0.272, 0.256, 0.234, 0.209,
     0.186, 0.162, 0.139, 0.112, 0.081, 0.046, 0.008, -0.032, -0.071, -0.110,
     -0.147, -0.180, -0.210, -0.235, -0.256, -0.275, -0.292, -0.307, -0.320,
     -0.332, -0.344, -0.355, -0.363, -0.367, -0.364, -0.351, -0.330, -0.299,
     -0.260, -0.217, -0.172, -0.128, -0.091, -0.060, -0.036, -0.022, -0.016,
     -0.020, -0.037, -0.065, -0.104, -0.151, -0.201, -0.253, -0.302, -0.347,
     -0.388, -0.426, -0.460, -0.491, -0.517, -0.539, -0.558, -0.575, -0.588,
     -0.600, -0.606, -0.607, -0.604, -0.598, -0.589, -0.577, -0.558, -0.531,
     -0.496, -0.454, -0.410, -0.364, -0.318, -0.276, -0.237, -0.203, -0.176,
     -0.157, -0.145, -0.142, -0.145, -0.154, -0.168, -0.185, -0.206, -0.230,
     -0.256, -0.286, -0.318, -0.351, -0.383, -0.414, -0.442, -0.467, -0.489,
     -0.508, -0.523, -0.535, -0.544, -0.552, -0.557, -0.560, -0.560, -0.557,
     -0.551, -0.542, -0.531, -0.519, -0.507, -0.494, -0.484, -0.476, -0.469,
     -0.463, -0.456, -0.449, -0.442, -0.435, -0.431, -0.429, -0.430, -0.435,
     -0.442, -0.452, -0.465, -0.479, -0.493, -0.506, -0.517, -0.526, -0.535,
     -0.548, -0.567, -0.592, -0.622, -0.655, -0.690, -0.728, -0.764, -0.795,
     -0.815, -0.823, -0.821]).reshape((-1, 1))

mp = MatrixProfile(subsequence_length=20, scale=False)
mp_series = mp.fit_transform([s_x])[0]
t_star = numpy.argmax(mp_series.ravel())

plt.figure()
ax = plt.subplot(2, 1, 1)  # First, raw time series
trans = mtransforms.blended_transform_factory(ax.transData, ax.transAxes)
plt.plot(s_x.ravel(), "b-")
plt.xlim([0, s_x.shape[0]])
plt.axvline(x=t_star,  c="red", linewidth=2)
plt.fill_between(x=[t_star, t_star+mp.subsequence_length], y1=0., y2=1.,
                 facecolor="r", alpha=.2, transform=trans)
plt.title("Raw time series")

plt.subplot(2, 1, 2)  # Second, Matrix Profile
plt.plot(mp_series.ravel(), "b-")
```

(continues on next page)

(continued from previous page)

```
plt.axvline(x=t_star,  c="red", linewidth=2, linestyle="dashed")
plt.xlim([0, s_x.shape[0]])
plt.title("Matrix Profile")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.148 seconds)

PAA and SAX features

This example presents a comparison between PAA[1], SAX[2] and 1d-SAX[3] features.

PAA (Piecewise Aggregate Approximation) corresponds to a downsampling of the original time series and, in each segment (segments have fixed size), the mean value is retained.

SAX (Symbolic Aggregate approXimation) builds upon PAA by quantizing the mean value. Quantization boundaries are computed for all symbols to be equiprobable, under a standard normal distribution assumption.

Finally, 1d-SAX is an extension of SAX in which each segment is represented by an affine function (2 parameters per segment are hence quantized: slope and mean value).

```
# 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))

# SAX transform
n_sax_symbols = 8
sax = SymbolicAggregateApproximation(n_segments=n_paa_segments,
                                     alphabet_size_avg=n_sax_symbols)
```

(continues on next page)

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[2] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[3] S. Malinowski, T. Guyet, R. Quiniou, R. Tavenard. 1d-SAX: a Novel Symbolic Representation for Time Series. IDA 2013.

(continued from previous page)

```
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 7.090 seconds)

Model Persistence

Many tslearn models can be saved to disk and used for predictions at a later time. This can be particularly useful when a model takes a long time to train.

Available formats: hdf5, json, pickle

Save a model to disk:

```
model.to_<format>
```

Load a model from disk:

```
model.from_<format>
```

Basic usage

```
# Instantiate a model
model = ModelClass(<hyper-parameters>)

# Train the model
model.fit(X_train)

# Save the model to disk
model.to_hdf5('./trained_model.hdf5')

# Load model from disk
model.from_hdf5('./trained_mode.hdf5')

# Make predictions
y = model.predict(X_test)
```

Note

For the following models the training data are saved to disk and may result in a large model file if the trainig dataset is large: `KNeighborsTimeSeries`, `KNeighborsTimeSeriesClassifier`, and `KernelKMeans`

```
0.008 --> 0.006 --> 0.005 --> 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 --> 0.003 --> 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 --> 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 --> 0.002 -->
↪ 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.002 --> 0.001␣
↪--> 0.001 --> 0.001 --> 0.001 --> 0.001 --> 0.001 --> 0.001 --> 0.001 -->
```

```
# Example using KShape
import os
import tempfile

import numpy
import matplotlib.pyplot as plt

from tslearn.clustering import KShape
from tslearn.datasets import CachedDatasets
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

seed = 0
```

(continues on next page)

(continued from previous page)

```
numpy.random.seed(seed)
X_train, y_train, X_test, y_test = CachedDatasets().load_dataset("Trace")

# Keep first 3 classes and 50 first time series
X_train = X_train[y_train < 4]
X_train = X_train[:50]
numpy.random.shuffle(X_train)
# For this method to operate properly, prior scaling is required
X_train = TimeSeriesScalerMeanVariance().fit_transform(X_train)
sz = X_train.shape[1]

# Instantiate k-Shape model
init=numpy.array([X_train[44], X_train[47], X_train[0]])
ks = KShape(n_clusters=3, verbose=True, init=init)

# Train
ks.fit(X_train)

with tempfile.TemporaryDirectory() as tmpdir:
    # Save model
    filename = os.path.join(tmpdir, "ks_trained.hdf5")
    if not os.path.isfile(filename):
        ks.to_hdf5(filename)

    # Load model
    trained_ks = KShape.from_hdf5(filename)

# Use loaded model to make predictions
y_pred = trained_ks.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 8.235 seconds)

CHAPTER

FIVE

CITING *TSLEARN*

If you use *tslearn* in a scientific publication, we would appreciate citations:

Bibtex entry:

```
@article{JMLR:v21:20-091,
  author  = {Romain Tavenard and Johann Faouzi and Gilles Vandewiele and
             Felix Divo and Guillaume Androz and Chester Holtz and
             Marie Payne and Roman Yurchak and Marc Ru{\ss}wurm and
             Kushal Kolar and Eli Woods},
  title   = {Tslearn, A Machine Learning Toolkit for Time Series Data},
  journal = {Journal of Machine Learning Research},
  year    = {2020},
  volume  = {21},
  number  = {118},
  pages   = {1-6},
  url     = {http://jmlr.org/papers/v21/20-091.html}
}
```

BIBLIOGRAPHY

[1] F. Petitjean, A. Ketterlin & P. Gancarski. A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition, Elsevier, 2011, Vol. 44, Num. 3, pp. 678-693

[2] D. Schultz and B. Jain. Nonsmooth Analysis and Subgradient Methods for Averaging in Dynamic Time Warping Spaces. Pattern Recognition, 74, 340-358.

[1] F. Petitjean, A. Ketterlin & P. Gancarski. A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition, Elsevier, 2011, Vol. 44, Num. 3, pp. 678-693

[2] D. Schultz and B. Jain. Nonsmooth Analysis and Subgradient Methods for Averaging in Dynamic Time Warping Spaces. Pattern Recognition, 74, 340-358.

[1] M. Cuturi, M. Blondel "Soft-DTW: a Differentiable Loss Function for Time-Series," ICML 2017.

[1] Kernel k-means, Spectral Clustering and Normalized Cuts. Inderjit S. Dhillon, Yuqiang Guan, Brian Kulis. KDD 2004.

[2] Fast Global Alignment Kernels. Marco Cuturi. ICML 2011.

[1] J. Paparrizos & L. Gravano. k-Shape: Efficient and Accurate Clustering of Time Series. SIGMOD 2015. pp. 1855-1870.

[1] Peter J. Rousseeuw (1987). "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis". Computational and Applied Mathematics 20: 53-65.

[2] Wikipedia entry on the Silhouette Coefficient

[1] A. Bagnall, J. Lines, W. Vickers and E. Keogh, The UEA & UCR Time Series Classification Repository, www.timeseriesclassification.com

[1] A. Bagnall, J. Lines, W. Vickers and E. Keogh, The UEA & UCR Time Series Classification Repository, www.timeseriesclassification.com

[1] A. Dachraoui, A. Bondu & A. Cornuejols. Early classification of time series as a non myopic sequential decision making problem. ECML/PKDD 2015

[1] C. M. Yeh, Y. Zhu, L. Ulanova, N.Begum et al. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. ICDM 2016.

[2] STUMPY documentation https://stumpy.readthedocs.io/en/latest/

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[1] 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] FRÉCHET, M. "Sur quelques points du calcul fonctionnel. Rendiconti del Circolo Mathematico di Palermo", 22, 1–74, 1906.

[2] 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] F. Zhou and F. Torre, "Canonical time warping for alignment of human behavior". NIPS 2009.

[1] F. Zhou and F. Torre, "Canonical time warping for alignment of human behavior". NIPS 2009.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[2] Z. Zhang, R. Tavenard, A. Bailly, X. Tang, P. Tang, T. Corpetti Dynamic time warping under limited warping path length. Information Sciences, vol. 393, pp. 91–107, 2017.

[1] H. Sakoe, S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 26(1), pp. 43–49, 1978.

[2] Z. Zhang, R. Tavenard, A. Bailly, X. Tang, P. Tang, T. Corpetti Dynamic time warping under limited warping path length. Information Sciences, vol. 393, pp. 91–107, 2017.

[1] M. Vlachos, D. Gunopoulos, and G. Kollios. 2002. "Discovering Similar Multidimensional Trajectories", In Proceedings of the 18th International Conference on Data Engineering (ICDE '02). IEEE Computer Society, USA, 673.

[1] M. Vlachos, D. Gunopoulos, and G. Kollios. 2002. "Discovering Similar Multidimensional Trajectories", In Proceedings of the 18th International Conference on Data Engineering (ICDE '02). IEEE Computer Society, USA, 673.

[1] M. Vlachos, D. Gunopoulos, and G. Kollios. 2002. "Discovering Similar Multidimensional Trajectories", In Proceedings of the 18th International Conference on Data Engineering (ICDE '02). IEEE Computer Society, USA, 673.

[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] Keogh, E. Exact indexing of dynamic time warping. In International Conference on Very Large Data Bases, 2002. pp 406-417.

[1] Keogh, E. Exact indexing of dynamic time warping. In International Conference on Very Large Data Bases, 2002. pp 406-417.

[1] M. Cuturi, "Fast global alignment kernels," ICML 2011.

[1] M. Cuturi, "Fast global alignment kernels," ICML 2011.

[1] Marco Cuturi & Mathieu Blondel. "Soft-DTW: a Differentiable Loss Function for Time-Series", ICML 2017.

[2] Mathieu Blondel, Arthur Mensch & Jean-Philippe Vert. "Differentiable divergences between time series", International Conference on Artificial Intelligence and Statistics, 2021.

[1] FRÉCHET, M. "Sur quelques points du calcul fonctionnel. Rendiconti del Circolo Mathematico di Palermo", 22, 1–74, 1906.

[2] 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] FRÉCHET, M. "Sur quelques points du calcul fonctionnel. Rendiconti del Circolo Mathematico di Palermo", 22, 1–74, 1906.

[2] 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] FRÉCHET, M. "Sur quelques points du calcul fonctionnel. Rendiconti del Circolo Mathematico di Palermo", 22, 1–74, 1906.

[2] 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] S. Malinowski, T. Guyet, R. Quiniou, R. Tavenard. 1d-SAX: a Novel Symbolic Representation for Time Series. IDA 2013.

[1] S. Malinowski, T. Guyet, R. Quiniou, R. Tavenard. 1d-SAX: a Novel Symbolic Representation for Time Series. IDA 2013.

[1] S. Malinowski, T. Guyet, R. Quiniou, R. Tavenard. 1d-SAX: a Novel Symbolic Representation for Time Series. IDA 2013.

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[1] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[1] E. Keogh & M. Pazzani. Scaling up dynamic time warping for datamining applications. SIGKDD 2000, pp. 285–289.

[1] J. Lin, E. Keogh, L. Wei, et al. Experiencing SAX: a novel symbolic representation of time series. Data Mining and Knowledge Discovery, 2007. vol. 15(107)

[1] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

[1] J. Grabocka et al. Learning Time-Series Shapelets. SIGKDD 2014.

PYTHON MODULE INDEX

t

INDEX

G

I

K

L

M

T

U