pyts Documentation

Release 0.7.0

Johann Faouzi

May 29, 2018

Contents

pyts is a Python package for time series transformation and classification. It aims to provide state-of-the-art as well as recently published algorithms for time series classification. Most of these algorithms transform time series, thus pyts provides several tools to perform these transformations.

CHAPTER 1

Installation

1.1 Dependencies

pyts has been tested on Python 2.7 and 3.5 with the following dependencies:

- numpy (>= 1.8.2)
- scipy (>= 0.13.3)
- scikit-learn (>=0.17.0)

To run the examples matplotlib is required (matplotlib >= 2.0.0 has been tested).

1.2 User installation

If you already have a working installation of numpy, scipy and scikit-learn, you can easily install pyts using `pip`:

```
pip install pyts
```

You can also get the latest version of pyts by cloning the repository:

```
git clone https://github.com/johannfaouzi/pyts.git
cd pyts
pip install .
```

CHAPTER 2

User guide

2.1 Introduction

Time series are very common data and classifying them can be of interest in a lot of fields. However standard machine learning algorithms for classification, like Logistic Regression, Support Vector Machine or K-Nearest Neighbors with usual metrics, don't work very well. To be more precise, these algorithms don't work well on **raw time series of real numbers**. Most algorithms developed recently have been focusing on *transforming* the raw time series before applying a standard machine learning classification algorithm.

In the following sections we'll present the algorithms implemented in pyts. If you want more information about the algorithms, you can have a look at the references and the *Examples* section.

2.2 Preprocessing

It is standard in machine learning to perform some preprocessing on raw data. Likewise it is standard to perform some preprocessing on time series. Implemented algorithms can be found in the `pyts.preprocessing` module.

Currently the only preprocessing tool implemented is **StandardScaler** It performs standardization (z-normalization) for each time series: the preprocessed time series all have zero mean and unit variance. It is implemented as `pyts.preprocessing.StandardScaler`.

2.3 Approximation

Time series can be of huge size or be very noisy. It can be useful to sum up the most important information of each time series. Implemented algorithms to approximate a time series can be found in the `pyts.approximation` module.

The first algorithm implemented is **Piecewise Aggregate Approximation (PAA)**. The main idea of this algorithm is to apply windows along a time series and to take the mean value in each window. It is implemented as `pyts.approximation.PAA`.

The second algorithm implemented is **Discrete Fourier Transform (DFT)**. The idea is to approximate a time series with a subsample of its Fourier coefficients. The selected Fourier coefficients are either the first ones (as they represent the trend of the time series) or the ones that discriminate the different classes the most if a vector of class labels is provided. It is implemented as `pyts.approximation.DFT`.

2.3.1 References

- Eamonn J. Keogh and Michael J. Pazzani. A simple dimensionality reduction technique for fast similarity search in large time series databases. *Knowledge Discovery and Data Mining* ,2000.
- Christos Faloutsos, M. Ranganathan and Yannis Manolopoulos. Fast Subsequence Matching in Time-Series Databases. *ACM SIGMOD Record*, 2000.

2.4 Quantization

One of the most interesting parts in time series classification is that several state-of-the-art algorithms use text mining techniques for classification and thus transform time series into bag of words. But first a time series of real numbers needs to be transformed into a sequence of letters. Implemented algorithms that quantize time series can be found in the `pyts.quantization` module.

The first algorithm implemented is **Symbolic Aggregate approXimation (SAX)**. For each time series, bins are computed using gaussian or empirical quantiles. Then each datapoint is replaced by the bin it is in. It is implemented as `pyts.quantization.SAX`.

The second algorithm implemented is **Multiple Coefficient Binning (MCB)**. The idea is very similar to SAX and the difference is that the quantization is applied at each timestamp. It is implemented as `pyts.quantization.MCB`.

The third algorithm implemented is **Symbolic Fourier Approximation (SFA)**. It performs DFT then MCB, i.e. MCB is applied to the selected Fourier coefficients of each time series. It is implemented as `pyts.quantization.SFA`.

2.4.1 References

- Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. Experiencing SAX: a Novel Symbolic Representation of Time Series. *Data Mining and Knowledge Discovery*, 2007.
- Patrick Schäfer and Mikael Högqvist. SFA: A Symbolic Fourier Approximation and Index for Similarity Search in High Dimensional Datasets. *ACM International Conference Proceeding Series*, 2012.

2.5 Bag of Words

Now that you know how you can transform a time series of real numbers into a sequence of letters, it's time to create bag of words. These algorithms are can be found in the `pyts.bow` module.

The only algorithm implemented for the moment is **Bag of Words (BOW)**. It applies a sliding window of fixed length along the sequence of letters to create words. It is implemented as `pyts.bow.BOW`.

2.6 Transformation

The `pyts.transformation` module consists of more complex algorithms that transform a dataset of raw time series with shape [n_samples, n_timestamps] into a more standard dataset of features with shape [n_samples, n_features] that can be used as input data for a standard machine learning classification algorithm.

The first algorithm implemented is **Bag-of-SFA Symbols (BOSS)**. Each time series is first transformed into a bag of words using SFA and BOW. After this transformation the features that are created are the frequencies of each word. It is implemented as `pyts.transformation.BOSS`.

The second algorithm implemented is **Word ExtrAction for time SEries cLassification (WEASEL)**. The idea is similar to BOSS: first transform each time series into a bag of words then compute the frequencies of each word. WEASEL is more sophisticated in the sense that the selected Fourier coefficients are the most discrimative ones (based on the one-way ANOVA test), several lengths for the sliding window are used and the most discriminative features (i.e. words) are kept (based on the chi-2 test). It is implemented as `pyts.transformation.WEASEL`.

2.6.1 References

- Patrick Schäfer. The BOSS is concerned with time series classification in the presence of noise. *Data Mining and Knowledge Discovery*, 2015.
- Patrick Schäfer and Ulf Leser. Fast and Accurate Time Series Classification with WEASEL. *CoRR*, 2017.

2.7 Classification

The `pyts.classification` module consists of several classification algorithms.

The first algorithm implemented is **K-Nearest Neighbors (KNN)**. For time series classification it is the go-to algorithm for a good baseline. The most common metrics used for time series classification are the Euclidean distance and the Dynamic Time Warping distance. It is implemented as `pyts.classification.KNNClassifier`.

The second algorithm implemented is **SAX-VSM**. The outline of this algorithm is to first transform raw time series into bags of words using SAX and BOW, then merge, for each class label, all bags of words for this class label into only one bag of words, and finally compute tf-idf for each bag of words. This leads to a tf-idf vector for each class label. To predict an unlabeled time series, this time series if first transformed into a term frequency vector, then the predicted label is the one giving the highest cosine similarity among the tf-idf vectors learned in the training phase. It is implemented as `pyts.classification.SAXVSMClassifier`.

The third algorithm implemented is **Bag-of-SFA Symbols in Vector Space (BOSSVS)**. The outline of this algorithm is quite similar to the one of SAX-VSM but words are created using SFA instead of SAX. It is implemented as `pyts.classification.BOSSVSClassifier`.

2.7.1 References

- Meinard Müller. Dynamic Time Warping (DTW). *Information Retrieval for Music and Motion*, 2007.

- Senin Pavel and Malinchik Sergey. SAX-VSM: Interpretable Time Series Classification Using SAX and Vector Space Model. *Data Mining (ICDM), 2013 IEEE 13th International Conference on, pp.1175,1180*, 2013.
- Patrick Schäfer. Scalable Time Series Classification. *DMKD* and *ECML/PKDD*, 2016.

2.8 Image

Instead of transforming a time series into a bag of words, it is also possible to transform it into an image ! The `pyts.image` module consists of several algorithms that perform that kind of transformation.

The first algorithm implemented is **Recurrence Plot**. It transforms a time series into a matrix where each value corresponds to the distance between two trajectories (a trajectory is a sub time series, i.e. a subsequence of back-to-back values of a time series). The matrix can be binarized using a threshold. It is implemented as `pyts.image.RecurrencePlots`.

The second algorithm implemented is **Gramian Angular Field (GAF)**. First a time series is represented as polar coordinates. Then the time series can be transformed into a **Gramian Angular Summation Field (GASF)** when the cosine of the sum of the angular coordinates is computed or a **Gramian Angular Difference Field (GADF)** when the sine of the difference of the angular coordinates is computed. It is implemented as `pyts.image.GASF` and `pyts.image.GADF`.

The third algorithm implemented is **Markov Transition Field (MTF)**. The outline of the algorithm is to first quantize a time series using SAX, then to compute the Markov transition matrix (the quantized time series is seen as a Markov chain) and finally to compute the Markov transition field from the transition matrix. It is implemented as `pyts.image.MTF`.

2.8.1 References

- J.-P. Eckmann, S. Oliffson Kamphorst and D. Ruelle. Recurrence Plots of Dynamical Systems. *Europhysics Letters*, 1987.
- Zhiguang Wang and Tim Oates. Imaging time-series to improve classification and imputation. *Proceedings of the 24th International Conference on Artificial Intelligence*, 2015.

2.9 Decomposition

The `pyts.decomposition` module consists of algorithms that decompose a time series into several time series. The idea is to distinguish the different parts of time series, such as the trend, the noise, etc.

The only algorithm implemented currently is **Singular Spectrum Analysis (SSA)**. The outline of the algorithm is to first compute a matrix from a time series using lagged vectors, then compute the eigenvalues and eigenvectors of this matrix multiplied by its transpose, after compute the eigenmatrices and finally compute the time series for each eigenmatrice. It is implemented as `pyts.decomposition.SSA`.

2.9.1 References

- Nina Golyandina and Anatoly Zhigljavsky. Singular Spectrum Analysis for Time Series. 2013

CHAPTER 3

API Documentation

Here you can find the API documentation for the different modules.

- approximation
- bow
- classification
- decomposition
- image
- preprocessing
- quantization
- transformation
- utils

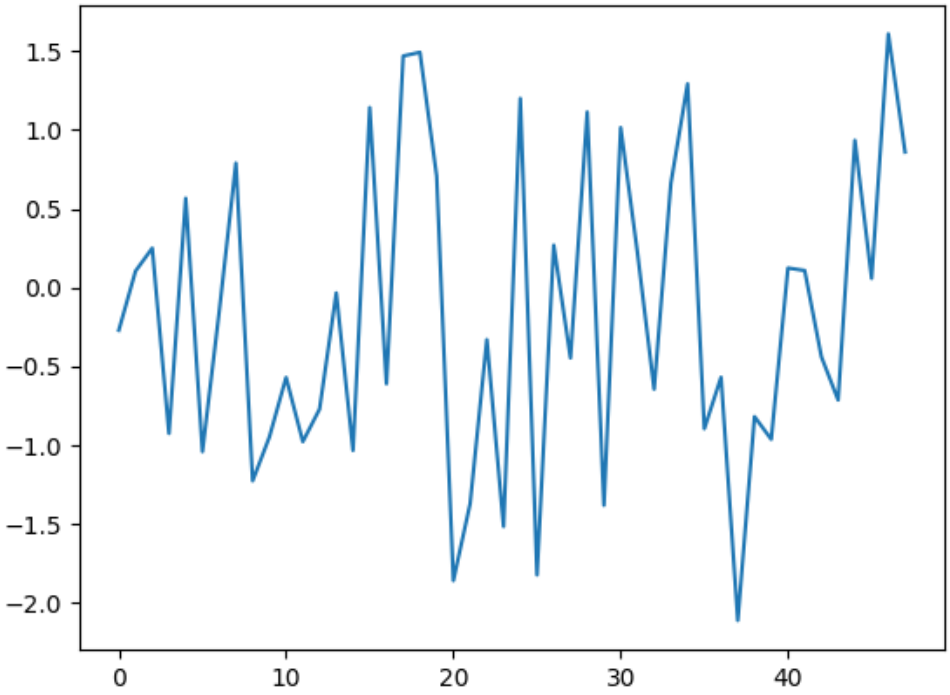
CHAPTER 4

Examples

Here you can find some introductory examples about the algorithms implemented in this package. It may help you understand how to use the implementations of these algorithms as well as what these algorithms do.

4.1 Plotting a time series

This example shows how you can plot a single time series.

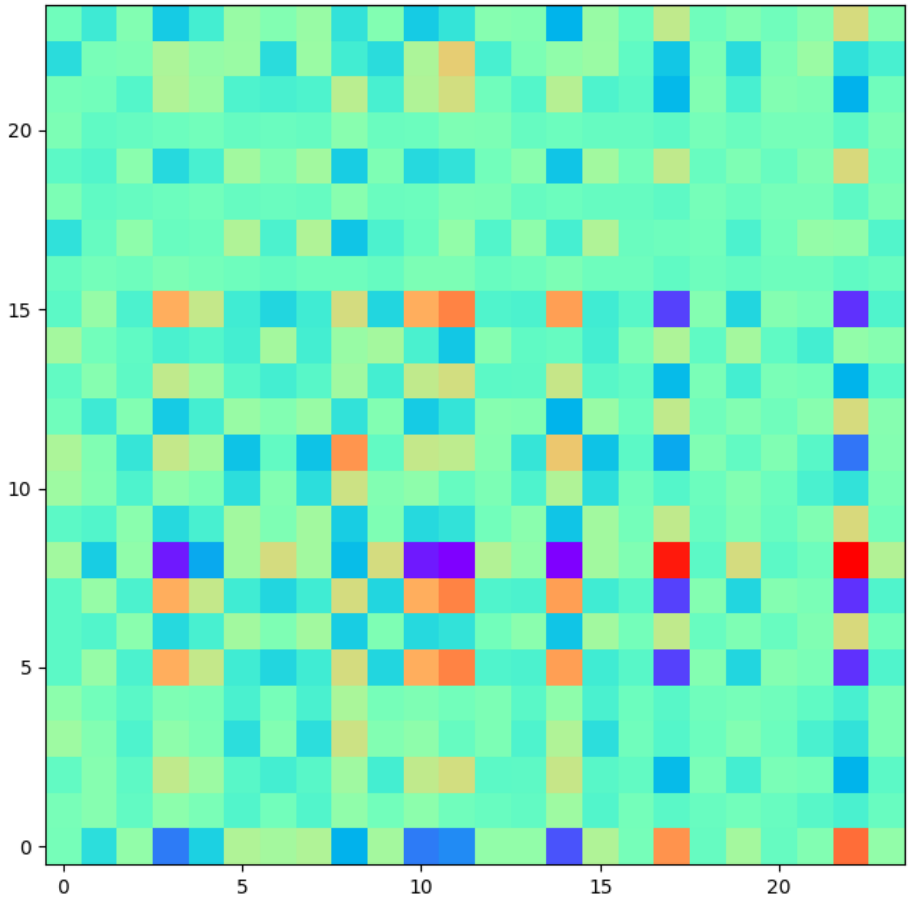



```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
n_samples, n_features = 100, 48

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# Plot the first time series
plt.plot(X[0])
plt.show()
```

Total running time of the script: (0 minutes 0.104 seconds)

4.2 Markov Transition Field

This example shows how you can transform a time series into a Markov Transition Field using `pyts.image.MTF`.




```
import numpy as np
import matplotlib.pyplot as plt
from pyts.image import MTF

# Parameters
n_samples, n_features = 100, 144

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# MTF transformation
image_size = 24
mtf = MTF(image_size)
X_mtf = mtf.fit_transform(X)

# Show the results for the first time series
plt.figure(figsize=(8, 8))
plt.imshow(X_mtf[0], cmap='rainbow', origin='lower')
```

(continues on next page)

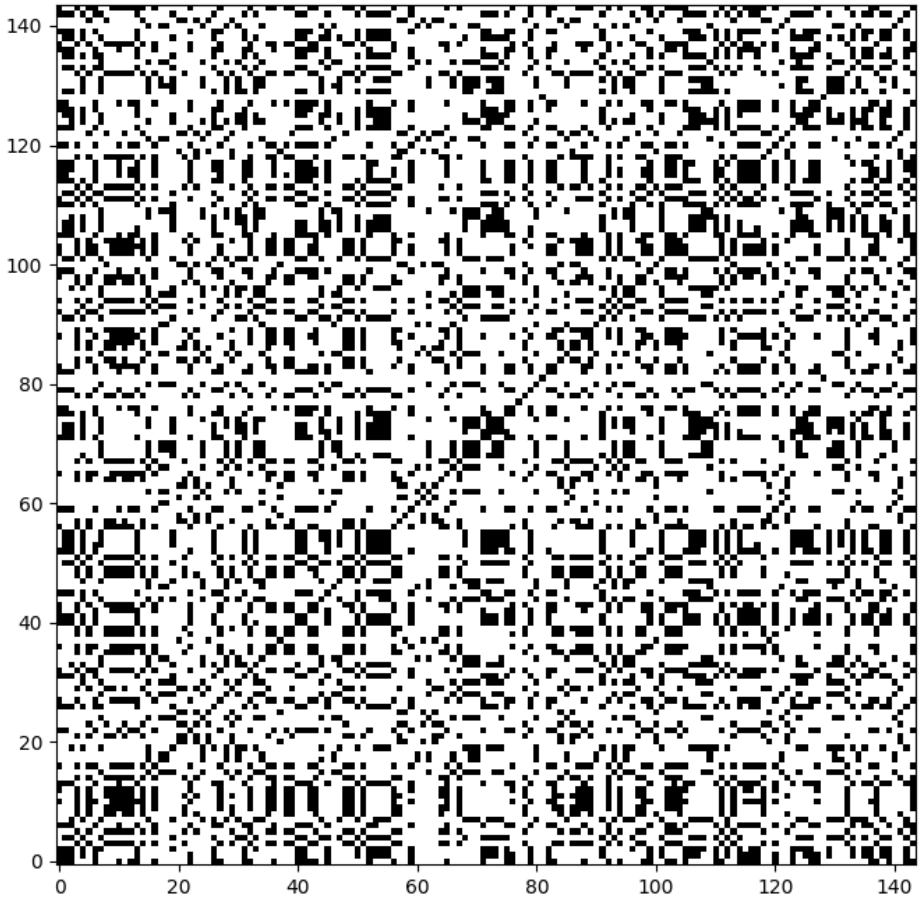
(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 2.686 seconds)

4.3 Recurrence Plots

This example shows how you can transform a time series into a Recurrence Plot using `pyts.image.RecurrencePlots`.




```
import numpy as np
import matplotlib.pyplot as plt
from pyts.image import RecurrencePlots

# Parameters
n_samples, n_features = 100, 144
```

(continues on next page)

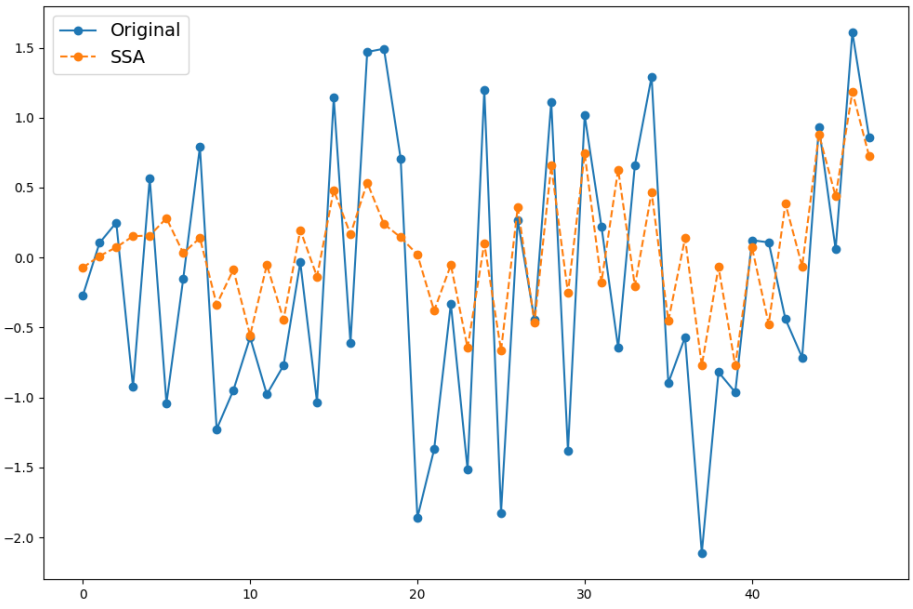
(continued from previous page)

```
# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# Recurrence plot transformation
rp = RecurrencePlots(dimension=1,
                     epsilon='percentage_points',
                     percentage=30)
X_rp = rp.fit_transform(X)

# Show the results for the first time series
plt.figure(figsize=(8, 8))
plt.imshow(X_rp[0], cmap='binary', origin='lower')
plt.show()
```

Total running time of the script: (0 minutes 0.411 seconds)

4.4 Singular Spectrum Analysis

This example shows how you can decompose a time series into several time series using `pyts.decomposition.SSA`.




```
import numpy as np
import matplotlib.pyplot as plt
from pyts.decomposition import SSA

# Parameters
```

(continues on next page)

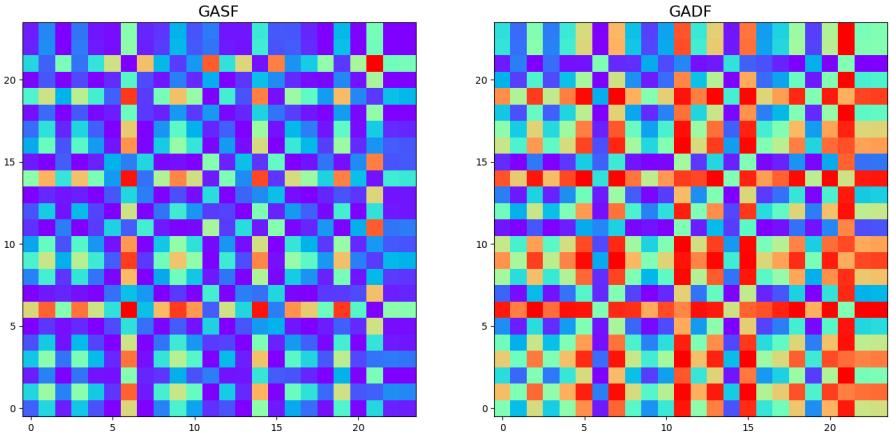
(continued from previous page)

```
n_samples, n_features = 100, 48

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# SSA transformation
window_size = 15
grouping = [[0, 1]]
ssa = SSA(window_size, grouping)
X_ssa = ssa.fit_transform(X)

# Show the results for the first time series
plt.figure(figsize=(12, 8))
plt.plot(X[0], 'o-', label='Original')
plt.plot(X_ssa[0, 0], 'o--', label='SSA')
plt.legend(loc='best', fontsize=14)
plt.show()
```

Total running time of the script: (0 minutes 2.087 seconds)

4.5 Gramian Angular Field

This example shows how you can transform a time series into a Gramian Angular Field using `pyts.image.GASF` for Gramian Angular Summation Field and `pyts.image.GADF` for Gramian Angular Difference Field.




```
import numpy as np
import matplotlib.pyplot as plt
from pyts.image import GASF, GADF

# Parameters
n_samples, n_features = 100, 144
```

(continues on next page)

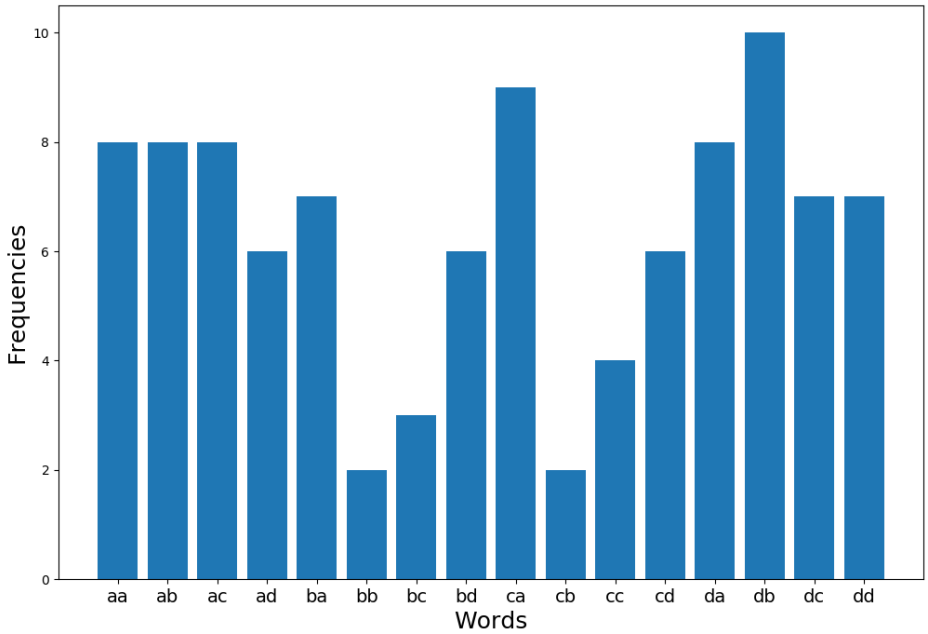
(continued from previous page)

```
# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# GAF transformations
image_size = 24
gasf = GASF(image_size)
X_gasf = gasf.fit_transform(X)
gadf = GADF(image_size)
X_gadf = gadf.fit_transform(X)

# Show the results for the first time series
plt.figure(figsize=(16, 8))
plt.subplot(121)
plt.imshow(X_gasf[0], cmap='rainbow', origin='lower')
plt.title("GASF", fontsize=16)
plt.subplot(122)
plt.imshow(X_gadf[0], cmap='rainbow', origin='lower')
plt.title("GADF", fontsize=16)
plt.show()
```

Total running time of the script: (0 minutes 0.324 seconds)

4.6 Bag-of-SFA Symbols

This example shows how the BOSS algorithm transforms a time series of real numbers into a sequence of frequencies of words. It is implemented as `pyts.transformation.BOSS`.

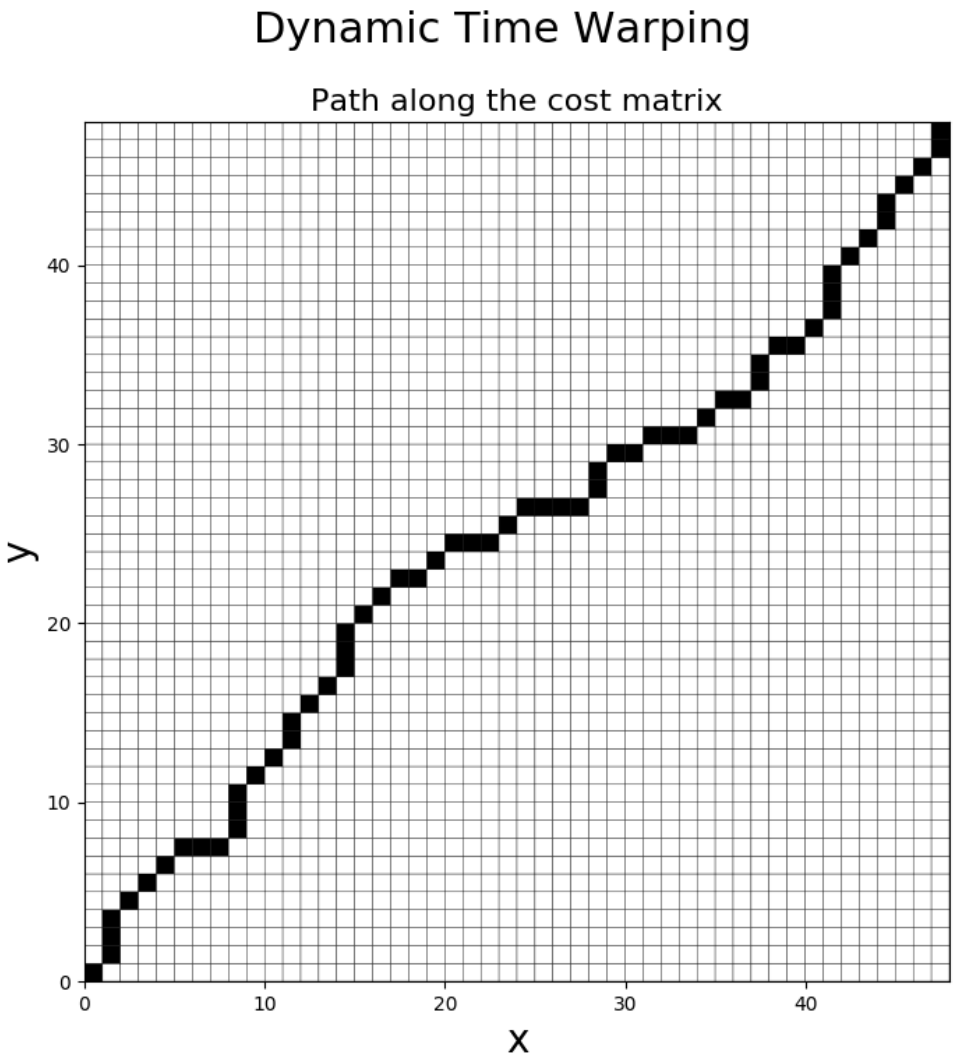


```
import numpy as np
import matplotlib.pyplot as plt
from pyts.transformation import BOSS

# Parameters
n_samples, n_features = 100, 144

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# BOSS transformation
boss = BOSS(n_coefs=2, window_size=12)
X_boss = boss.fit_transform(X).toarray()

# Visualize the transformation for the first time series
plt.figure(figsize=(12, 8))
plt.bar(np.arange(X_boss[0].size), X_boss[0])
plt.xticks(np.arange(X_boss[0].size),
           np.vectorize(boss.vocabulary_.get)(np.arange(X_boss[0].size)),
           fontsize=14)
plt.xlabel("Words", fontsize=18)
plt.ylabel("Frequencies", fontsize=18)
plt.show()
```

Total running time of the script: (0 minutes 0.480 seconds)

4.7 Dynamic Time Warping

This example shows how to compute and visualize the optimal path when computing the Dynamic Time Warping distance between two time series. It is implemented as `pyts.utils.dtw()`.




```
import numpy as np
import matplotlib.pyplot as plt
from pyts.utils import dtw

# Parameters
n_samples, n_features = 2, 48

# Toy dataset
rng = np.random.RandomState(41)
x, y = rng.randn(n_samples, n_features)

# Dynamic Time Warping
D, path = dtw(x, y, dist='absolute', return_path=True)

# Visualize the result
timestamps = np.arange(n_features + 1)
matrix = np.zeros([n_features + 1, n_features + 1])
for i in range(len(path)):
    matrix[path[i][0], path[i][1]] = 1
```

(continues on next page)

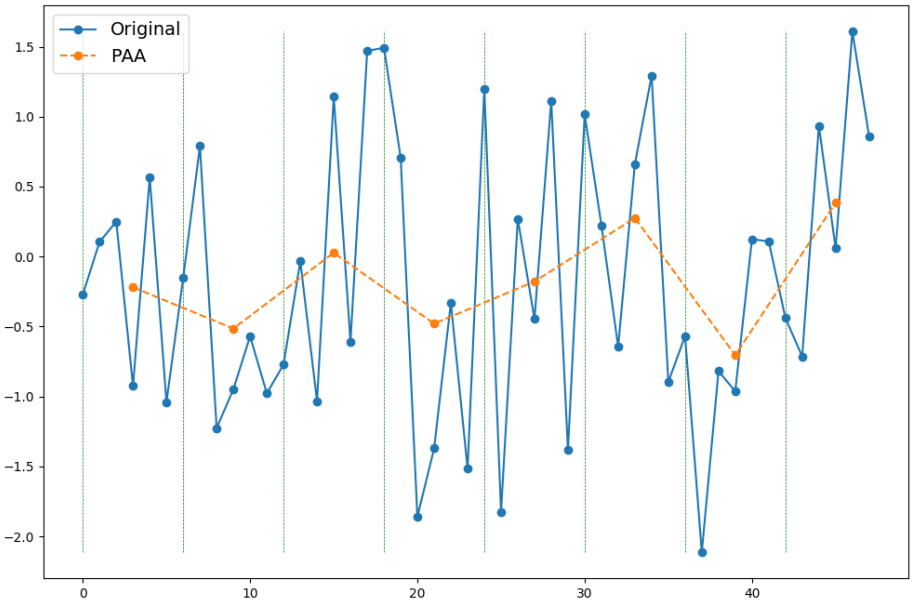
(continued from previous page)

```
plt.figure(figsize=(8, 8))
plt.pcolor(timestamps, timestamps, matrix, edgecolors='k', cmap='Greys')
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.title("Path along the cost matrix", fontsize=16)
plt.suptitle("Dynamic Time Warping", fontsize=22)
plt.show()
```

Total running time of the script: (0 minutes 0.282 seconds)

4.8 Piecewise Aggregate Approximation

This example shows how you can approximate a time series using `pyts.approximation.PAA`.




```
import numpy as np
import matplotlib.pyplot as plt
from pyts.approximation import PAA

# Parameters
n_samples, n_features = 100, 48

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# PAA transformation
window_size = 6
```

(continues on next page)

(continued from previous page)

```
paa = PAA(window_size=window_size)
X_paa = paa.transform(X)

# Show the results for the first time series
plt.figure(figsize=(12, 8))
plt.plot(np.arange(n_features), X[0], 'o-', label='Original')
plt.plot(np.arange(window_size // 2,
                   n_features + window_size // 2,
                   window_size), X_paa[0], 'o--', label='PAA')
plt.vlines(np.arange(0, n_features, window_size),
           X[0].min(), X[0].max(), color='g', linestyles='--', linewidth=0.5)
plt.legend(loc='best', fontsize=14)
plt.show()
```

Total running time of the script: (0 minutes 0.104 seconds)

4.9 Bag of Words

This example shows how you can transform a quantized time series (i.e. a time series represented as a sequence of letters) into a bag of words using `pyts.bow.BOW`.

Out:

```
Original time series:
['a' 'd' 'a' 'c' 'a' 'b' 'd' 'b' 'd' 'b' 'c' 'b' 'a' 'a' 'd' 'd' 'c' 'c'
 'a' 'a' 'c' 'b' 'b' 'd' 'a' 'a' 'd' 'a' 'a' 'd']


Bag of words without numerosity reduction:
{a, d, a, c, a, b, d, b, d, b, c, b, a, a, d, d, c, c, a, a, c, b, b, d, a,␣
↪a, d, a, a, d}


Bag of words with numerosity reduction:
{a, d, a, c, a, b, d, b, d, b, c, b, a, d, c, a, c, b, d, a, d, a, d}
```

```
import numpy as np
from pyts.bow import BOW

# Parameters
n_samples = 100
n_features = 30
n_bins = 4
window_size = 1
alphabet = np.array([chr(i) for i in range(97, 97 + n_bins)])

# Toy dataset
rng = np.random.RandomState(41)
X = alphabet[rng.randint(n_bins, size=(n_samples, n_features))]
```

(continues on next page)

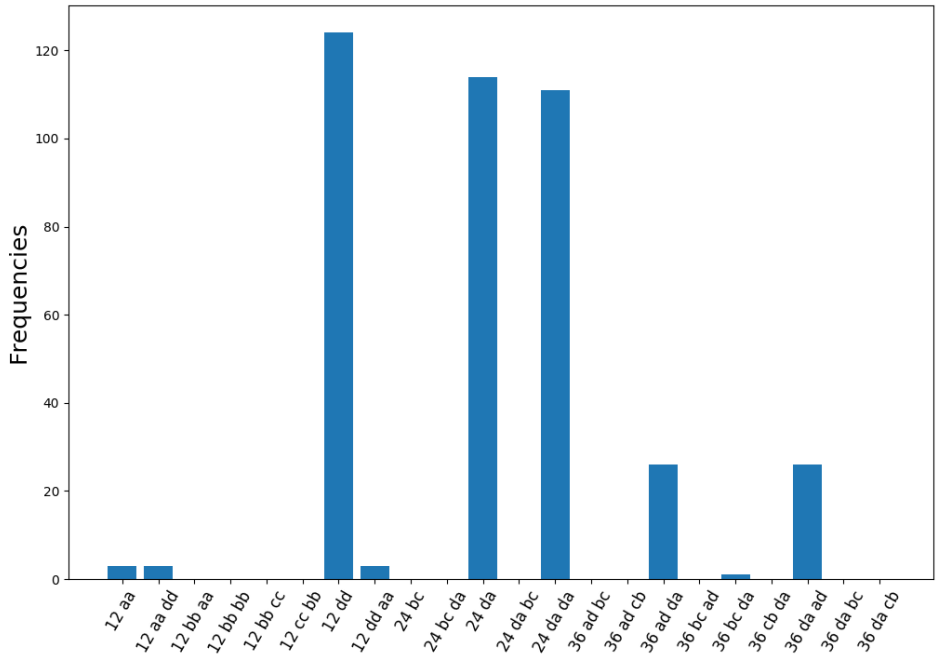
(continued from previous page)

```
# Bag-of-words transformation
bow = BOW(window_size, numerosity_reduction=False)
X_bow = bow.fit_transform(X)
bow_num = BOW(window_size, numerosity_reduction=True)
X_bow_num = bow_num.fit_transform(X)

print("Original time series:")
print(X[0])
print("\n")
print("Bag of words without numerosity reduction:")
print(''.join(["{", X_bow[0].replace(" ", ", "), "}"]))
print("\n")
print("Bag of words with numerosity reduction:")
print(''.join(["{", X_bow_num[0].replace(" ", ", "), "}"]))
```

Total running time of the script: (0 minutes 0.123 seconds)

4.10 Word ExtrAction for time SEries cLassification

This example shows how the WEASEL algorithm transforms a time series of real numbers into a sequence of frequencies of words. It is implemented as `pyts.transformation.WEASEL`.




```
import numpy as np
import matplotlib.pyplot as plt
from pyts.transformation import WEASEL

# Parameters
```

(continues on next page)

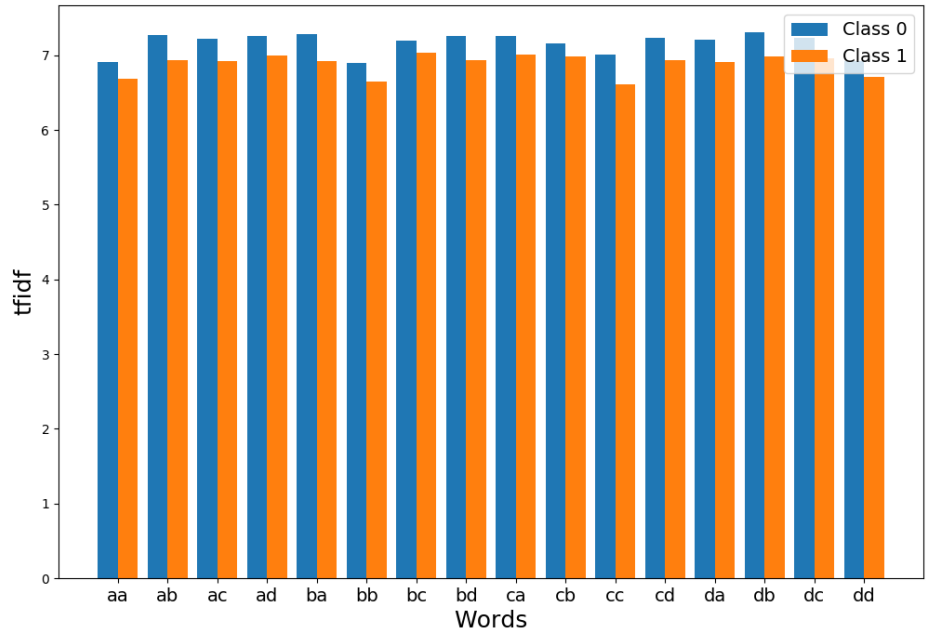
(continued from previous page)

```
n_samples, n_features = 100, 144
n_classes = 2

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)
y = rng.randint(n_classes, size=n_samples)

# WEASEL transformation
weasel = WEASEL(n_coefs=2, window_sizes=[12, 24, 36], pvalue_threshold=0.2)
X_weasel = weasel.fit_transform(X, y).toarray()

# Visualize the transformation for the first time series
plt.figure(figsize=(12, 8))
plt.bar(np.arange(X_weasel[0].size), X_weasel[0])
plt.xticks(np.arange(X_weasel[0].size),
           np.vectorize(weasel.vocabulary_.get)(np.arange(X_weasel[0].size)),
           fontsize=12, rotation=60)
plt.xlabel("Words", fontsize=18)
plt.ylabel("Frequencies", fontsize=18)
plt.show()
```

Total running time of the script: (0 minutes 10.130 seconds)

4.11 SAXVSM

This example shows how the SAXVSM algorithm transforms a dataset consisting of time series and their corresponding labels into a document-term matrix using tfidf. Each class is represented as a tfidf vector. For an unlabeled time series, the predicted label is the label of the tfidf vector giving the highest cosine similarity with the tf vector of the unlabeled time series. Here we plot the tfidf vectors for each clss. SAXVSM algorithm is implemented as `pyts.classification.SAXVSMClassifier`.

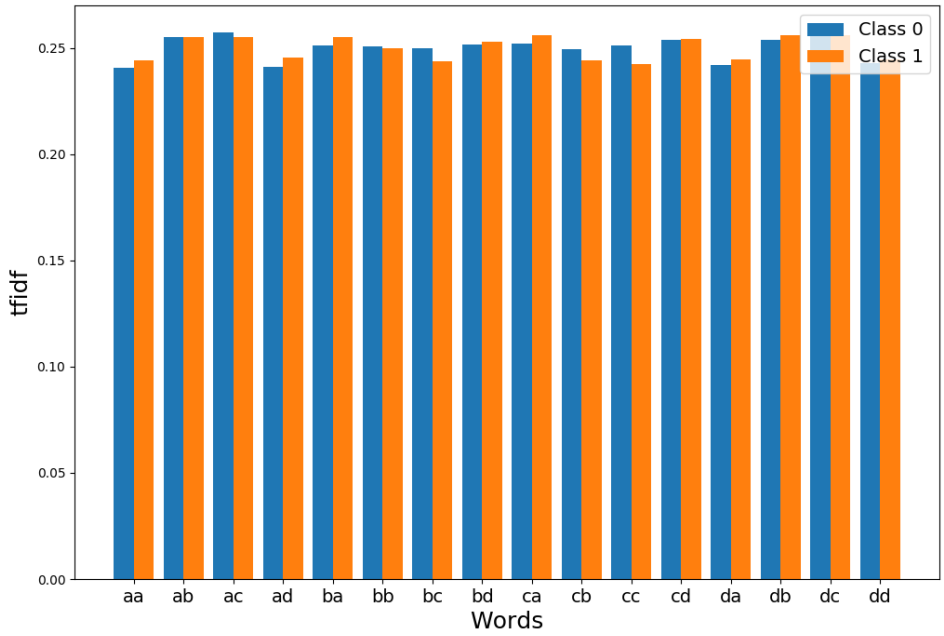



```
import numpy as np
import matplotlib.pyplot as plt
from pyts.classification import SAXVSMClassifier

# Parameters
n_samples, n_features = 100, 144
n_classes = 2

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)
y = rng.randint(n_classes, size=n_samples)

# SAXVSM transformation
saxvsm = SAXVSMClassifier(window_size=2, sublinear_tf=True)
saxvsm.fit(X, y)
tfidf = saxvsm.tfidf_.toarray()

# Visualize the transformation
plt.figure(figsize=(12, 8))
plt.bar(np.arange(tfidf[0].size) - 0.2, tfidf[0], width=0.4, label='Class 0')
plt.bar(np.arange(tfidf[0].size) + 0.2, tfidf[1], width=0.4, label='Class 1')
plt.xticks(np.arange(tfidf[0].size),
           np.vectorize(saxvsm.vocabulary_.get)(np.arange(tfidf[0].size)),
           fontsize=14)
plt.xlabel("Words", fontsize=18)
plt.ylabel("tfidf", fontsize=18)
plt.legend(loc='best', fontsize=14)
plt.show()
```

Total running time of the script: (0 minutes 0.606 seconds)

4.12 Bag-of-SFA Symbols in Vector Space

This example shows how the BOSSVS algorithm transforms a dataset consisting of time series and their corresponding labels into a document-term matrix using tfidf. Each class is represented as a tfidf vector. For an unlabeled time series, the predicted label is the label of the tfidf vector giving the highest cosine similarity with the tf vector of the unlabeled time series. Here we plot the tfidf vectors for each clss. BOSSVS algorithm is implemented as `pyts.classification.BOSSVSClassifier`.




```
import numpy as np
import matplotlib.pyplot as plt
from pyts.classification import BOSSVSClassifier

# Parameters
n_samples, n_features = 100, 144
n_classes = 2

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)
y = rng.randint(n_classes, size=n_samples)

# BOSSVS transformation
bossvs = BOSSVSClassifier(n_coefs=2, window_size=24, variance_selection=True)
bossvs.fit(X, y)
tfidf = bossvs.tfidf_.toarray()

# Visualize the transformation
plt.figure(figsize=(12, 8))
plt.bar(np.arange(tfidf[0].size) - 0.2, tfidf[0], width=0.4, label='Class 0')
plt.bar(np.arange(tfidf[0].size) + 0.2, tfidf[1], width=0.4, label='Class 1')
```

(continues on next page)

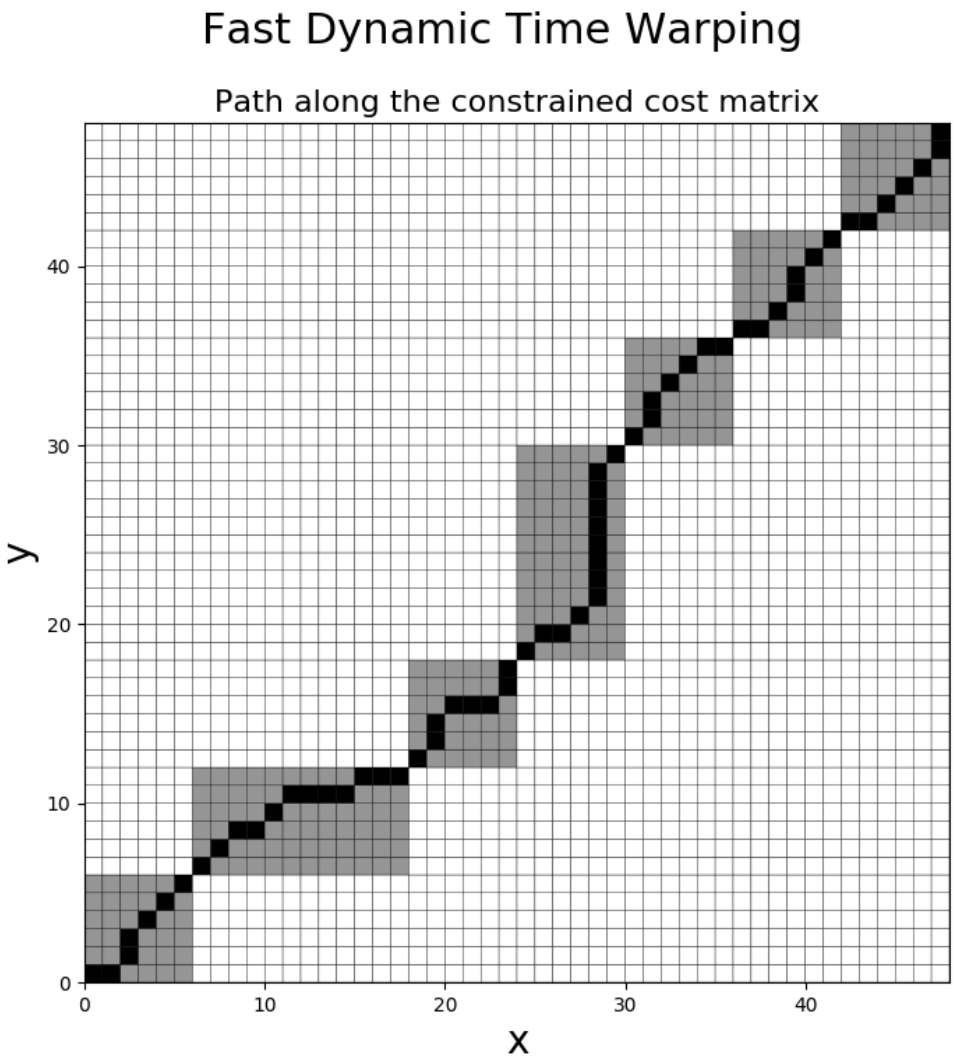
(continued from previous page)

```
plt.xticks(np.arange(tfidf[0].size),
           np.vectorize(bossvs.vocabulary_.get)(np.arange(tfidf[0].size)),
           fontsize=14)
plt.xlabel("Words", fontsize=18)
plt.ylabel("tfidf", fontsize=18)
plt.legend(loc='best', fontsize=14)
plt.show()
```

Total running time of the script: (0 minutes 0.578 seconds)

4.13 Fast Dynamic Time Warping

This example shows how to compute and visualize the optimal path when computing the Fast Dynamic Time Warping distance between two time series. It is implemented as `pyts.utils.fast_dtw()`.

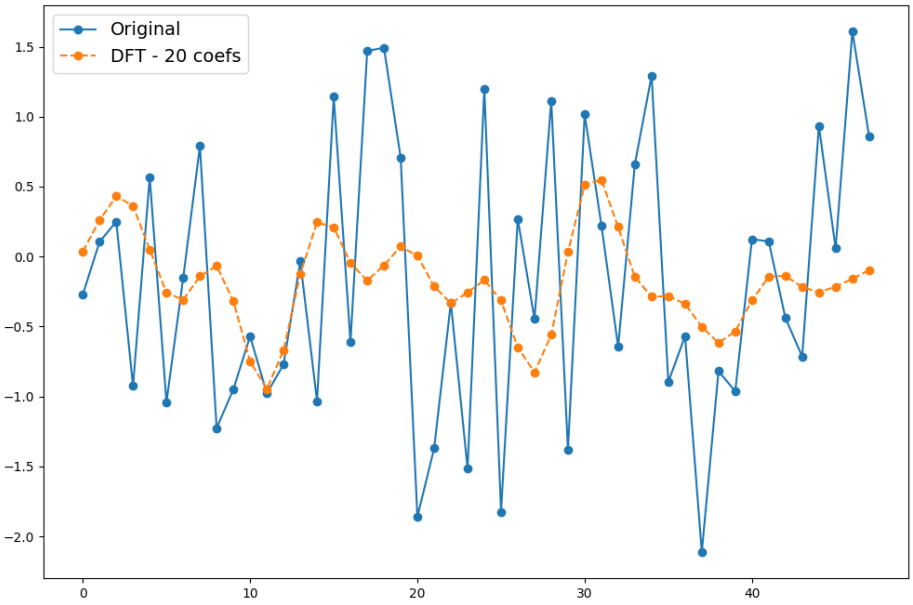


```
import numpy as np
import matplotlib.pyplot as plt
from pyts.utils import fast_dtw

# Parameters
n_samples, n_features = 2, 48

# Toy dataset
rng = np.random.RandomState(41)
x, y = rng.randn(n_samples, n_features)

# Dynamic Time Warping
region, D, path = fast_dtw(x, y, dist='absolute', window_size=6,
                           approximation=False, return_path=True)

# Visualize the result
timestamps = np.arange(n_features + 1)
matrix = np.zeros([n_features + 1, n_features + 1])
for i in range(n_features):
    for j in region[i]:
        matrix[j, i] = 0.5
for i in range(len(path)):
    matrix[path[i][0], path[i][1]] = 1

plt.figure(figsize=(8, 8))
plt.pcolor(timestamps, timestamps, matrix, edgecolors='k', cmap='Greys')
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.title("Path along the constrained cost matrix", fontsize=16)
plt.suptitle("Fast Dynamic Time Warping", fontsize=22)
plt.show()
```

Total running time of the script: (0 minutes 0.342 seconds)

4.14 Discrete Fourier Transform

This example shows how you can approximate a time series using only some of its Fourier coefficients using `pyts.approximation.DFT`.




```
import numpy as np
import matplotlib.pyplot as plt
from pyts.approximation import DFT

# Parameters
n_samples, n_features = 100, 48

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# DFT transformation
n_coefs = 20
norm_mean = False
norm_std = False
dft = DFT(n_coefs=n_coefs, norm_mean=norm_mean, norm_std=norm_std)
X_dft = dft.fit_transform(X)

# Compute the approximation for the first time series
timestamps = np.arange(n_features) / n_features
x_dft = np.zeros(n_features)
for n in range(n_coefs // 2):
    x_dft += X_dft[0, 2 * n] * np.cos(2 * n * np.pi * timestamps) / n_
↪features
    x_dft += X_dft[0, (2 * n) + 1] * np.sin(2 * n * np.pi * timestamps) / \
        n_features

# Show the results for the first time series
plt.figure(figsize=(12, 8))
plt.plot(X[0], 'o-', label='Original')
plt.plot(x_dft, 'o--', label='DFT - {0} coefs'.format(n_coefs))
```

(continues on next page)

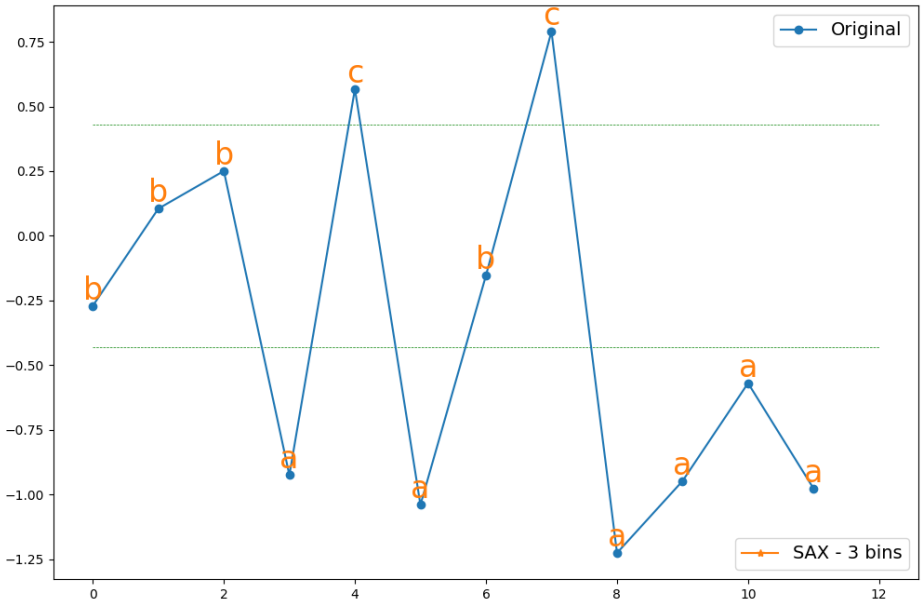
(continued from previous page)

```
plt.legend(loc='best', fontsize=14)
plt.show()
```

Total running time of the script: (0 minutes 0.060 seconds)

4.15 Symbolic Aggregate approXimation

This example shows how you can quantize a time series (i.e. transform a sequence of real numbers into a sequence of letters) using `pyts.quantization.SAX`.




```
import numpy as np
import matplotlib.lines as mlines
import matplotlib.pyplot as plt
from scipy.stats import norm
from pyts.quantization import SAX

# Parameters
n_samples, n_features = 100, 12

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# SAX transformation
n_bins = 3
quantiles = 'gaussian'
sax = SAX(n_bins=n_bins, quantiles=quantiles)
X_sax = sax.fit_transform(X)
```

(continues on next page)

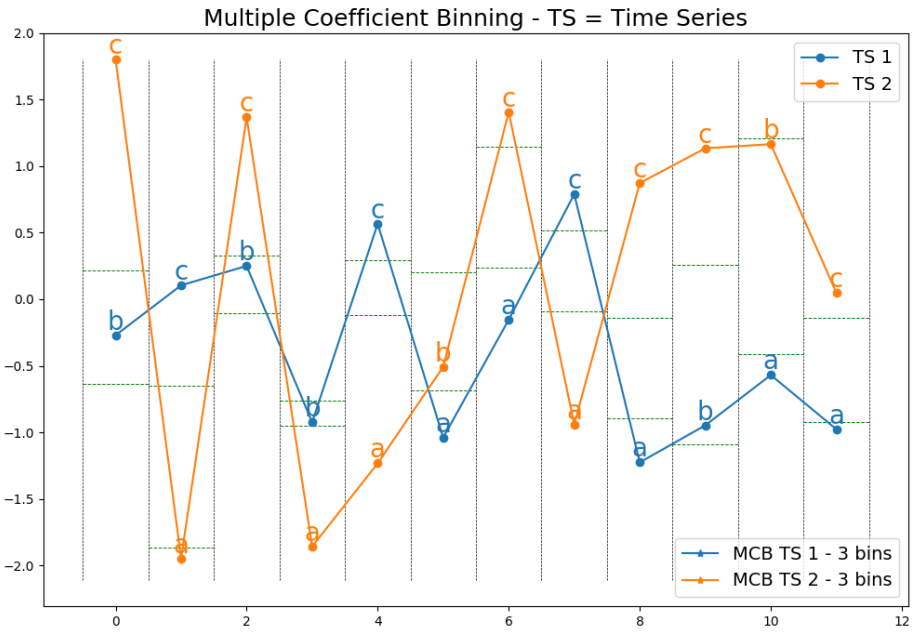
(continued from previous page)

```
# Compute gaussian bins
bins = norm.ppf(np.linspace(0, 1, n_bins + 1)[1:-1])

# Show the results for the first time series
plt.figure(figsize=(12, 8))
plt.plot(X[0], 'o-', label='Original')
for x, y, s in zip(range(n_features), X[0], X_sax[0]):
    plt.text(x, y, s, ha='center', va='bottom', fontsize=24, color='#ff7f0e')
plt.hlines(bins, 0, n_features, color='g', linestyles='--', linewidth=0.5)
sax_legend = mlines.Line2D([], [], color='#ff7f0e', marker='*',
                           label='SAX - {0} bins'.format(n_bins))
first_legend = plt.legend(handles=[sax_legend], fontsize=14, loc=4)
ax = plt.gca().add_artist(first_legend)
plt.legend(loc='best', fontsize=14)
plt.show()
```

Total running time of the script: (0 minutes 0.063 seconds)

4.16 Multiple Coefficient Binning

This example shows how the MCB algorithm transforms a dataset of time series of real numbers into a list of sequences of letters. It is implemented as `pyts.quantization.MCB`.




```
import numpy as np
import matplotlib.lines as mlines
import matplotlib.pyplot as plt
from pyts.quantization import MCB
```

(continues on next page)

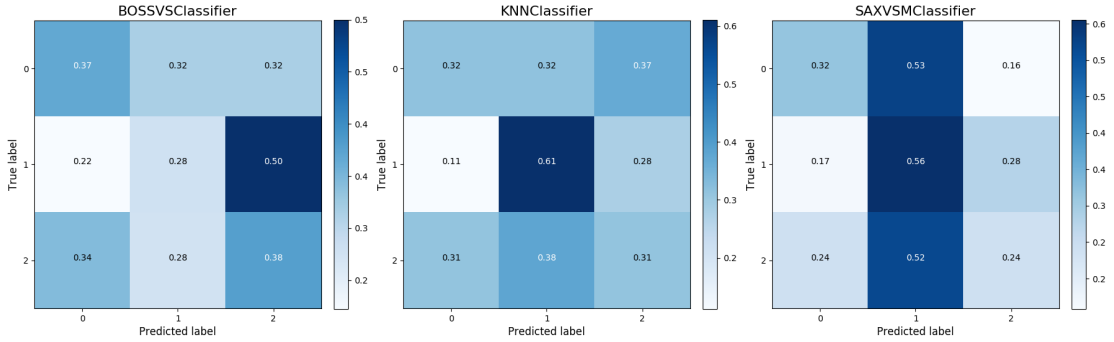
(continued from previous page)

```
# Parameters
n_samples, n_features = 6, 12

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)

# MCB transformation
n_bins = 3
quantiles = 'empirical'
mcb = MCB(n_bins=n_bins, quantiles=quantiles)
X_mcb = mcb.fit_transform(X)

# Compute bins
bins = mcb._bins

# Show the results for the first time series
plt.figure(figsize=(12, 8))
# First time series
plt.plot(X[0], 'o-', label='TS 1')
for x, y, s in zip(range(n_features), X[0], X_mcb[0]):
    plt.text(x, y, s, ha='center', va='bottom', fontsize=20, color='#1f77b4')
# Second time series
plt.plot(X[5], 'o-', label='TS 2')
for x, y, s in zip(range(n_features), X[5], X_mcb[5]):
    plt.text(x, y, s, ha='center', va='bottom', fontsize=20, color='#ff7f0e')
plt.hlines(bins, np.arange(n_features) - 0.5, np.arange(n_features) + 0.5,
           color='g', linestyles='--', linewidth=0.7)
plt.vlines(np.arange(n_features + 1) - 0.5, X.min(), X.max(),
           linestyles='--', linewidth=0.5)
mcb_legend_1 = mlines.Line2D([], [], color='#1f77b4', marker='*',
                             label='MCB TS 1 - {0} bins'.format(n_bins))
mcb_legend_2 = mlines.Line2D([], [], color='#ff7f0e', marker='*',
                             label='MCB TS 2 - {0} bins'.format(n_bins))
first_legend = plt.legend(handles=[mcb_legend_1, mcb_legend_2],
                          fontsize=14, loc=4)
ax = plt.gca().add_artist(first_legend)
plt.legend(loc='best', fontsize=14)
plt.title("Multiple Coefficient Binning - TS = Time Series", fontsize=18)
plt.show()
```

Total running time of the script: (0 minutes 0.087 seconds)

4.17 Classifiers

This example shows how to use the classifiers from `pyts.classification`. If you are familiar with scikit-learn classifiers, it is straightforward. The confusion matrix for each classifier is also plotted.




```
import numpy as np
import matplotlib.pyplot as plt
from itertools import product
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from pyts.classification import (BOSSVSClassifier, SAXVSMClassifier,
                                 KNNClassifier)

# Parameters
n_samples, n_features = 200, 144
n_classes = 3

# Toy dataset
rng = np.random.RandomState(41)
X = rng.randn(n_samples, n_features)
y = rng.randint(n_classes, size=n_samples)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=4141)

# BOSSVSClassifier
bossvs = BOSSVSClassifier(n_coefs=4, window_size=24)
bossvs.fit(X_train, y_train)
y_pred_boss = bossvs.predict(X_test)

# SAXVSMClassifier
saxvsm = SAXVSMClassifier()
saxvsm.fit(X_train, y_train)
y_pred_saxvsm = saxvsm.predict(X_test)

# KNNClassifier
knn = KNNClassifier()
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)

# Confusion matrices
cm_boss = confusion_matrix(y_test, y_pred_boss)
cm_boss = cm_boss.astype('float') / cm_boss.sum(axis=1)[:, np.newaxis]

cm_saxvsm = confusion_matrix(y_test, y_pred_saxvsm)
cm_saxvsm = cm_saxvsm.astype('float') / cm_saxvsm.sum(axis=1)[:, np.newaxis]

cm_knn = confusion_matrix(y_test, y_pred_knn)
```

(continues on next page)

(continued from previous page)

```
cm_knn = cm_knn.astype('float') / cm_knn.sum(axis=1)[:, np.newaxis]

dictionary = {"BOSSVSClassifier": cm_boss,
              "SAXVSMClassifier": cm_saxvsm,
              "KNNClassifier": cm_knn}

# Plot confusion matrices
plt.figure(figsize=(18, 6))
for idx, (title, cm) in zip(range(1, 4), dictionary.items()):
    plt.subplot(130 + idx)
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    thresh = (cm.max() + cm.min()) / 2.
    for i, j in product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], '0.2f'),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.colorbar(fraction=0.046, pad=0.04, format='%.1f')
    plt.title(title, fontsize=16)
    tick_marks = np.arange(n_classes)
    plt.xticks(tick_marks, tick_marks)
    plt.yticks(tick_marks, tick_marks)
    plt.tight_layout()
    plt.ylabel('True label', fontsize=12)
    plt.xlabel('Predicted label', fontsize=12)
plt.show()
```

Total running time of the script: (0 minutes 2.705 seconds)

CHAPTER 5

Citation

pyts is registered on Zenodo. If you use it in a scientific publication, please cite us:

```
@misc{johann_faouzi_2018_1244152,
author       = {Johann Faouzi},
title        = {{pyts: a Python package for time series transformation and
 ↪classification}},
month        = may,
year         = 2018,
doi          = {10.5281/zenodo.1244152},
url          = {https://doi.org/10.5281/zenodo.1244152}
            }
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search