

---

**sequentia**

*Release 0.6.0*

**Jan 05, 2020**



<b>1</b>	<b>Changelog</b>	<b>3</b>
<b>2</b>	<b>Hidden Markov Model (HMM)</b>	<b>7</b>
<b>3</b>	<b>Ensemble Hidden Markov Model Classifier (HMMClassifier)</b>	<b>11</b>
<b>4</b>	<b>Dynamic Time Warping <math>k</math>-Nearest Neighbors Classifier (DTWKNN)</b>	<b>15</b>
<b>5</b>	<b>Zero-trimming (<code>trim_zeros</code>)</b>	<b>19</b>
<b>6</b>	<b>Centering (<code>center</code>)</b>	<b>21</b>
<b>7</b>	<b>Standardizing (<code>standardize</code>)</b>	<b>23</b>
<b>8</b>	<b>Downsampling (<code>downsample</code>)</b>	<b>25</b>
<b>9</b>	<b>Filtering (<code>filtrate</code>)</b>	<b>27</b>
<b>10</b>	<b>Discrete Fourier Transform (<code>fft</code>)</b>	<b>29</b>
<b>11</b>	<b>Combined Preprocessing (<code>Preprocess</code>)</b>	<b>31</b>
<b>12</b>	<b>Documentation Search and Index</b>	<b>33</b>
	<b>Python Module Index</b>	<b>35</b>
	<b>Index</b>	<b>37</b>



# sequentia

Sequentia is a collection of machine learning algorithms for performing the classification of isolated temporal sequences.

Each isolated sequence is generally modeled as a section of a longer multivariate time series that represents the entire sequence. Naturally, this fits the description of many types of problems such as:

- isolated word utterance frequencies in speech audio signals,
- isolated hand-written character pen-tip trajectories,
- isolated hand or head gestures positions in a video or motion-capture recording.

Most modern machine learning algorithms won't work directly out of the box when applied to such sequential data – mostly due to the fact that the dependencies between observations at different time frames must be considered, and also because each isolated sequence generally has a different duration.

Sequentia offers some appropriate classification algorithms for these kinds of tasks.



### 1.1 0.6.0

#### 1.1.1 Major changes

- Add package tests and Travis CI support. (#56)
- Remove Python v3.8+ support. (#56)
- Rename `normalize` preprocessing method to `center`, since it just centers an observation sequence. (#62)
- Add `standardize` preprocessing method for standardizing (standard scaling) an observation sequence. (#63)
- Add `trim_zeros` preprocessing method for removing zero-observations from an observation sequence. (#67)

#### 1.1.2 Minor changes

- *(Internal)* Add `Validator.random_state` for validating random state objects and seeds. (#56)
- *(Internal)* Internalize `Validator` and `topology` (`Topology`, `ErgodicTopology`, `LeftRightTopology`) classes. (#57)
- *(Internal)* Use proper documentation format for topology classes. (#58)

### 1.2 0.5.0

#### 1.2.1 Major changes

- Add `Preprocess.summary()` to display an ordered summary of preprocessing transformations. (#54)
- Add mean and median filtering preprocessing methods. (#48)
- Use median filtering and decimation downsampling by default. (#52)

- Modify preprocessing boundary conditions (#51):
  - Use a bi-directional window for filtering to resolve boundary problems.
  - Modify downsampling method to downsample residual observations.

## 1.2.2 Minor changes

- Add supported topologies (left-right and ergodic) to feature list. (#53)
- Add restrictions on preprocessing parameters: downsample factor and window size. (#50)
- Allow `Preprocess` class to be used to apply preprocessing transformations to a single observation sequence. (#49)

## 1.3 0.4.0

### 1.3.1 Major changes

- Re-add `euclidean` metric as `DTWKNN` default. (#43)

### 1.3.2 Minor changes

- Add explicit labels to `evaluate()` in `HMMClassifier` example. (#44)

## 1.4 0.3.0

### 1.4.1 Major changes

- Add proper documentation, hosted on [Read The Docs](#). (#40, #41)

## 1.5 0.2.0

### 1.5.1 Major changes

- Add multi-processing support for `DTWKNN` predictions. (#29)
- Rename the `fit_transform()` function in `Preprocess` to `transform()` since there is nothing being fitted. (#35)
- (*Internal*) Modify package classifiers in `setup.py` (#31):
  - Set development status classifier to `Pre-Alpha`.
  - Add Python version classifiers for `v3.5+`.
  - Specify UNIX and macOS operating system classifiers.



## 1.5.2 Minor changes

- *(Internal)* Finish tutorial and example notebooks. (#35)
- *(Internal)* Rename `examples` directory to `notebooks`. (#32)
- *(Internal)* Host notebooks statically on `nbviewer`. (#32)
- *(Internal)* Add reference to Pomegranate paper and repository. (#30)
- *(Internal)* Add badges to `README.md`. (#28)

## 1.6 0.1.0

### 1.6.1 Major changes

Nothing, initial release!

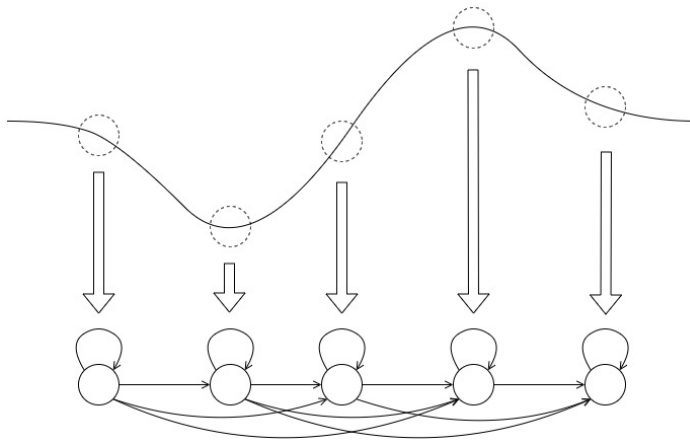


---

## Hidden Markov Model (HMM)

---

The **Hidden Markov Model (HMM)** is a state-based statistical model that can be used to represent an individual observation sequence class  $c$ . As seen in the diagram below, the rough idea is that each state should correspond to one ‘section’ of the sequence.



A single HMM is modeled by the `HMM` class.

### 2.1 Parameters and Training

The ‘sections’ in the image above are determined by the parameters of the HMM, explained below.

- **Initial state distribution  $\pi$ :**  
A discrete probability distribution that dictates the probability of the HMM starting in each state.
- **Transition probability matrix  $A$ :**  
A matrix whose rows represent a discrete probability distribution that dictates how likely the HMM is to transition to each state, given some current state.
- **Emission probability distributions  $B$ :**

A collection of  $N$  continuous multivariate probability distributions (one for each state) that each dictate the probability of the HMM generating an observation  $\mathbf{o}$ , given some current state. Recall that we are generally considering multivariate observation sequences – that is, at time  $t$ , we have an observation  $\mathbf{o}^{(t)} = (o_1^{(t)}, o_2^{(t)}, \dots, o_D^{(t)})$ . The fact that the observations are multivariate necessitates a multivariate emission distribution. Sequentia uses the [multivariate Gaussian distribution](#).

In order to learn these parameters, we must train the HMM on examples that are labeled with the class  $c$  that the HMM models. Denote the HMM that models class  $c$  as  $\lambda_c = (\pi_c, A_c, B_c)$ . We can use the [Baum-Welch algorithm](#) (an application of the [Expectation-Maximization algorithm](#)) to fit  $\lambda_c$  and learn its parameters. This fitting is implemented by the `fit()` function.

## 2.1.1 Model Topologies

As we usually wish to preserve the natural ordering of time, we normally want to prevent our HMM from transitioning to previous states (this is shown in the figure above). This restriction leads to what known as a **left-right** HMM, and is the most commonly used type of HMM for sequential modeling. Mathematically, a left-right HMM is defined by an upper-triangular transition matrix.

If we allow transitions to any state at any time, this HMM topology is known as **ergodic**.

**Note:** Ergodicity is mathematically defined as having a transition matrix with no non-zero entries. Using the ergodic topology in Sequentia will still permit zero entries in the transition matrix, but will issue a warning stating that those probabilities will not be learned.

Sequentia offers both topologies, specified by a string parameter `topology` in the HMM constructor that takes values `'left-right'` or `'ergodic'`.

## 2.2 Making Predictions

A score for how likely a HMM is to generate an observation sequence is given by the [Forward algorithm](#). It calculates the likelihood  $\mathbb{P}(O|\lambda_c)$  of the HMM  $\lambda_c$  generating the observation sequence  $O$ .

**Note:** The likelihood does not account for the fact that a particular observation class may occur more or less frequently than other observation classes. Once an ensemble of HMM objects (represented by a `HMMClassifier`) is created and configured, this can be accounted for by calculating the joint probability (or un-normalized posterior)  $\mathbb{P}(O, \lambda_c) = \mathbb{P}(O|\lambda_c)\mathbb{P}(\lambda_c)$  and using this score to classify instead. The addition of the prior term  $\mathbb{P}(\lambda_c)$  accounts for some classes occurring more frequently than others.

## 2.3 Example

```

1 import numpy as np
2 from sequentia.classifiers import HMM
3
4 # Create some sample data
5 X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
6
7 # Create and fit a left-right HMM with random transitions and initial state_
  ↳distribution
8 hmm = HMM(label='class1', n_states=5, topology='left-right')
9 hmm.set_random_initial()
10 hmm.set_random_transitions()
11 hmm.fit(X)

```

For more elaborate examples, please have a look at the [example notebooks](#).

## 2.4 API reference

**class** `sequentia.classifiers.hmm.HMM` (*label*, *n\_states*, *topology='left-right'*, *random\_state=None*)

A hidden Markov model representing an isolated temporal sequence class.

### Parameters

**label: str** A label for the model, corresponding to the class being represented.

**n\_states: int** The number of states for the model.

**topology: {'ergodic', 'left-right'}** The topology for the model.

**random\_state: numpy.random.RandomState, int, optional** A random state object or seed for reproducible randomness.

### Attributes

**label: str** The label for the model.

**n\_states: int** The number of states for the model.

**n\_seqs: int** The number of observation sequences use to train the model.

**initial: numpy.ndarray** The initial state distribution of the model.

**transitions: numpy.ndarray** The transition matrix of the model.

**set\_uniform\_initial** (*self*)

Sets a uniform initial state distribution.

**set\_random\_initial** (*self*)

Sets a random initial state distribution.

**set\_uniform\_transitions** (*self*)

Sets a uniform transition matrix according to the topology.

**set\_random\_transitions** (*self*)

Sets a random transition matrix according to the topology.

**fit** (*self*, *X*, *n\_jobs=1*)

Fits the HMM to observation sequences assumed to be labeled as the class that the model represents.

### Parameters

**X: List[numpy.ndarray]** Collection of multivariate observation sequences, each of shape  $(T \times D)$  where  $T$  may vary per observation sequence.

**n\_jobs: int**

The number of jobs to run in parallel.

Setting this to -1 will use all available CPU cores.

**forward** (*self*, *sequence*)

Runs the forward algorithm to calculate the (negative log) likelihood of the model generating an observation sequence.

### Parameters

**sequence: numpy.ndarray** An individual sequence of observations of size  $(T \times D)$  where  $T$  is the number of time frames (or observations) and  $D$  is the number of features.

### Returns

**negative log-likelihood: float** The negative log-likelihood of the model generating the observation sequence.

---

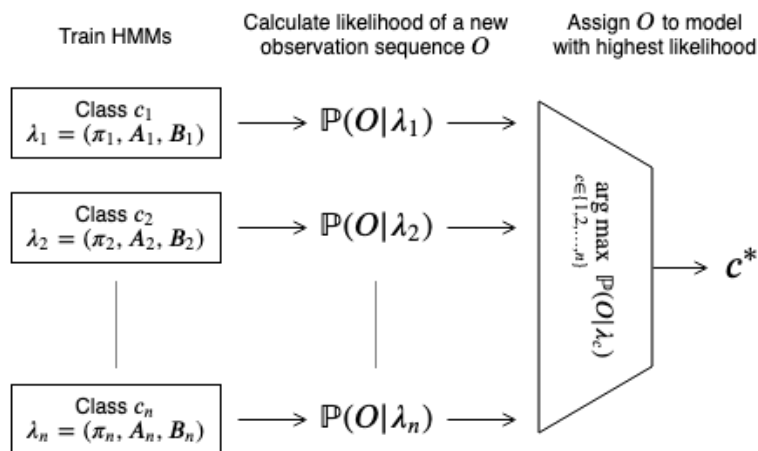
## Ensemble Hidden Markov Model Classifier (`HMMClassifier`)

---

Multiple HMMs can be combined to form an ensemble multi-class classifier. To classify a new observation sequence  $O'$ , this works by:

1. Creating and training the HMMs  $\lambda_1, \lambda_2, \dots, \lambda_N$ .
2. Calculating the likelihoods  $\mathbb{P}(O'|\lambda_1), \mathbb{P}(O'|\lambda_2), \dots, \mathbb{P}(O'|\lambda_N)$  of each model generating  $O'$ .  
**Note:** You can also use the un-normalized posterior  $\mathbb{P}(O'|\lambda_c)\mathbb{P}(\lambda_c)$  instead of the likelihood.
3. Choose the class represented by the HMM with the highest likelihood – that is,  
 $c^* = \arg \max_{c \in \{1, \dots, N\}} \mathbb{P}(O'|\lambda_c)$ .

These steps are summarized in the diagram below.



## 3.1 Example

```

1 import numpy as np
2 from sequentia.classifiers import HMM, HMMClassifier
3
4 # Set of possible labels
5 labels = ['class{}'.format(i) for i in range(5)]
6
7 # Create and fit some sample HMMs
8 hmms = []
9 for i, label in enumerate(labels):
10     hmm = HMM(label=label, n_states=(i + 3), topology='left-right')
11     hmm.set_random_initial()
12     hmm.set_random_transitions()
13     hmm.fit([np.arange((i + j * 20) * 30).reshape(-1, 3) for j in range(1, 4)])
14     hmms.append(hmm)
15
16 # Create some sample test data and labels
17 X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
18 y = ['class0', 'class1', 'class1']
19
20 # Create a classifier and calculate predictions and evaluations
21 clf = HMMClassifier()
22 clf.fit(hmms)
23 predictions = clf.predict(X)
24 accuracy, confusion = clf.evaluate(X, y, labels=labels)

```

For more elaborate examples, please have a look at the [example notebooks](#).

## 3.2 API reference

**class** `sequentia.classifiers.hmm.HMMClassifier`

An ensemble classifier that combines individual [HMM](#) objects, which model isolated sequences from different classes.

**fit** (*self*, *models*)

Fits the ensemble classifier with a collection of [HMM](#) objects.

### Parameters

**models:** `List[HMM]` or `Dict[Any, HMM]` A collection of [HMM](#) objects to use for classification.

**predict** (*self*, *X*, *prior=True*, *return\_scores=False*)

Predicts the label for an observation sequence (or multiple sequences) according to maximum likelihood or posterior scores.

### Parameters

**X:** `numpy.ndarray` or `List[numpy.ndarray]` An individual observation sequence or a list of multiple observation sequences.

**prior:** `bool` Whether to calculate a prior for each model and perform MAP estimation by scoring with the joint probability (or un-normalized posterior)  $\mathbb{P}(O, \lambda_c) = \mathbb{P}(O|\lambda_c)\mathbb{P}(\lambda_c)$ .

If this parameter is set to false, then the negative log likelihoods  $\mathbb{P}(O|\lambda_c)$  generated from the models' `forward()` function are used.



**return\_scores: bool** Whether to return the scores of each model on the observation sequence(s).

#### Returns

**prediction(s): str or List[str]** The predicted label(s) for the observation sequence(s).

If `return_scores` is true, then for each observation sequence, a tuple (*label*, *scores*) is returned for each label, consisting of the *scores* of each HMM and the *label* of the HMM with the best score.

**evaluate** (*self*, *X*, *y*, *prior=True*, *labels=None*)

Evaluates the performance of the classifier on a batch of observation sequences and their labels.

#### Parameters

**X: List[numpy.ndarray]** A list of multiple observation sequences.

**y: List[str]** A list of labels for the observation sequences.

**prior: bool** Whether to calculate a prior for each model and perform MAP estimation by scoring with the joint probability (or un-normalized posterior)  $\mathbb{P}(O, \lambda_c) = \mathbb{P}(O|\lambda_c)\mathbb{P}(\lambda_c)$ .

If this parameter is set to false, then the negative log likelihoods  $\mathbb{P}(O|\lambda_c)$  generated from the models' `forward()` function are used.

**labels: List[str]** A list of labels for ordering the axes of the confusion matrix.

#### Returns

**accuracy: float** The categorical accuracy of the classifier on the observation sequences.

**confusion: numpy.ndarray** The confusion matrix representing the discrepancy between predicted and actual labels.



---

## Dynamic Time Warping $k$ -Nearest Neighbors Classifier (DTWKNN)

---

Recall that the isolated sequences we are dealing with are represented as multivariate time series of different durations.

Suppose that our sequences are all  $D$ -dimensional. The main requirement of  $k$ -Nearest Neighbor ( $k$ -NN) classifiers is that each example must have the same number of dimensions – and hence, be in the same feature space. This is indeed the case with our  $D$ -dimensional sequences. However, we can't use  $k$ -NN with simple distance metrics such as Euclidean distance because we are comparing sequences (which represent an ordered collection of points in  $D$ -dimensional space) rather than individual points in  $D$ -dimensional space.

One distance metric that allows us to compare multivariate sequences of different length is [Dynamic Time Warping](#). Coupling this metric with  $k$ -NN creates a powerful classifier that assigns the class of a new observation sequence by looking at the classes of observation sequences with similar patterns.

However,  $k$ -NN classifiers suffer from the fact that they are non-parametric, which means that when predicting the class for a new observation sequence, we must look back at every observation sequence that was used to fit the model. To speed up prediction times, we have chosen to use a constrained DTW algorithm that sacrifices accuracy by calculating an approximate distance, but saves **a lot** of time. This is the [FastDTW](#) implementation, which has a *radius* parameter for controlling the imposed constraint on the distance calculation.

This approximate DTW  $k$ -NN classifier is implemented by the DTWKNN class.

### 4.1 Example

```
1 import numpy as np
2 from sequentia.classifiers import DTWKNN
3
4 # Create some sample data
5 X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
6 y = ['class0', 'class1', 'class1']
7
8 # Create and fit the classifier
```

(continues on next page)

```

9 clf = DTWKNN(k=1, radius=5)
10 clf.fit(X, y)
11
12 # Predict labels for the training data (just as an example)
13 clf.predict(X)

```

For more elaborate examples, please have a look at the [example notebooks](#).

## 4.2 API reference

**class** `sequentia.classifiers.dtwknn.DTWKNN` (*k*, *radius*, *metric*=<function euclidean>)  
 A k-Nearest Neighbor classifier that compares differing length observation sequences using the efficient Fast-DTW dynamic time warping algorithm.

### Parameters

**k: int** Number of neighbors.

**radius: int** Radius parameter for FastDTW.

**See:** Stan Salvador, and Philip Chan. “FastDTW: Toward accurate dynamic time warping in linear time and space.” *Intelligent Data Analysis* 11.5 (2007), 561-580.

**metric: callable** Distance metric for FastDTW.

**fit** (*self*, *X*, *y*)

Fits the classifier by adding labeled training observation sequences.

### Parameters

**X: List[numpy.ndarray]** A list of multiple observation sequences.

**y: List[str]** A list of labels for the observation sequences.

**predict** (*self*, *X*, *verbose*=*True*, *n\_jobs*=*1*)

Predicts the label for an observation sequence (or multiple sequences).

### Parameters

**X: numpy.ndarray or List[numpy.ndarray]** An individual observation sequence or a list of multiple observation sequences.

**verbose: bool** Whether to display a progress bar or not.

**n\_jobs: int**

The number of jobs to run in parallel.

Setting this to -1 will use all available CPU cores.

### Returns

**prediction(s): str or List[str]** The predicted label(s) for the observation sequence(s).

**evaluate** (*self*, *X*, *y*, *labels*=*None*, *verbose*=*True*, *n\_jobs*=*1*)

Evaluates the performance of the classifier on a batch of observation sequences and their labels.

### Parameters

**X: List[numpy.ndarray]** A list of multiple observation sequences.

**y: List[str]** A list of labels for the observation sequences.

**labels: List[str]** A list of labels for ordering the axes of the confusion matrix.

**verbose: bool** Whether to display a progress bar for predictions or not.

**n\_jobs: int**

The number of jobs to run in parallel.

Setting this to -1 will use all available CPU cores.

#### Returns

**accuracy: float** The categorical accuracy of the classifier on the observation sequences.

**confusion: numpy.ndarray** The confusion matrix representing the discrepancy between predicted and actual labels.



---

## Zero-trimming (`trim_zeros`)

---

Removes zero-observations from an observation sequence.

Many datasets consisting of sequential data often pad observation sequences with zeros in order to ensure that the machine learning algorithms receive sequences of equal length. Although this comes with the advantage of being able to represent the sequences in a matrix, the added zeros may affect the performance of the machine learning algorithms.

As the algorithms implemented by `Sequential` focus on supporting variable-length sequences out of the box, zero padding is not necessary, and can be removed with this method.

**NOTE:** This preprocessing method does not only remove trailing zeros from the start or end of a sequence, but will also remove **any** zero-observations that occur anywhere in the sequence.

For further information, please see the [preprocessing tutorial notebook](#).

### 5.1 Example

```
1 import numpy as np
2 from sequentia.preprocessing import trim_zeros
3
4 # Create some sample data
5 z = np.zeros((4, 3))
6 x = lambda i: np.vstack((z, np.random.random((10 * i, 3))), z)
7 X = [x(i) for i in range(1, 4)]
8
9 # Zero-trim the data
10 X = trim_zeros(X)
```

### 5.2 API reference

`sequentia.preprocessing.trim_zeros(X)`

Trim zero-observations from the input observation sequence(s).

**Parameters**

**X: numpy.ndarray or List[numpy.ndarray]** An individual observation sequence or a list of multiple observation sequences.

**Returns**

**trimmed: numpy.ndarray or List[numpy.ndarray]** The zero-trimmed input observation sequence(s).



---

## Centering (`center`)

---

Centers an observation sequence about the mean of its observations – that is, given:

$$O = \begin{pmatrix} o_1^{(1)} & o_2^{(1)} & \cdots & o_D^{(1)} \\ o_1^{(2)} & o_2^{(2)} & \cdots & o_D^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ o_1^{(T)} & o_2^{(T)} & \cdots & o_D^{(T)} \end{pmatrix} \quad \boldsymbol{\mu} = (\bar{o}_1 \quad \bar{o}_2 \quad \cdots \quad \bar{o}_D)$$

Where  $\bar{o}_d$  represents the mean of the  $d^{\text{th}}$  feature of  $O$ .

We subtract  $\boldsymbol{\mu}$  from each observation, or row in  $O$ . This centers the observations.

For further information, please see the [preprocessing tutorial notebook](#).

### 6.1 Example

```

1 import numpy as np
2 from sequentia.preprocessing import center
3
4 # Create some sample data
5 X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
6
7 # Center the data
8 X = center(X)

```

### 6.2 API reference

`sequentia.preprocessing.center(X)`

Centers an observation sequence (or multiple sequences) by centering observations around the mean.

#### Parameters

**X:** `numpy.ndarray` or `List[numpy.ndarray]` An individual observation sequence or a list of multiple observation sequences.

**Returns**

**centered:** `numpy.ndarray` or `List[numpy.ndarray]` The centered input observation sequence(s).

---

## Standardizing (`standardize`)

---

Standardizes an observation sequence by transforming observations so that they have zero mean and unit variance. For further information, please see the [preprocessing tutorial notebook](#).

### 7.1 Example

```
1 import numpy as np
2 from sequentia.preprocessing import standardize
3
4 # Create some sample data
5 X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
6
7 # Standardize the data
8 X = standardize(X)
```

### 7.2 API reference

`sequentia.preprocessing.standardize(X)`

Standardizes an observation sequence (or multiple sequences) by transforming observations so that they have zero mean and unit variance.

#### Parameters

**X:** `numpy.ndarray` or `List[numpy.ndarray]` An individual observation sequence or a list of multiple observation sequences.

#### Returns

**standardized:** `numpy.ndarray` or `List[numpy.ndarray]` The standardized input observation sequence(s).



---

## Downsampling (downsample)

---

Downsampling reduces the number of frames in an observation sequence according to a specified downsample factor and one of two methods: **averaging** and **decimation**.

This is an especially helpful preprocessing method for speeding up classification times.

For further information, please see the [preprocessing tutorial notebook](#).

### 8.1 Example

```
1 import numpy as np
2 from sequentia.preprocessing import downsample
3
4 # Create some sample data
5 X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
6
7 # Downsample the data with downsample factor 5 and decimation
8 X = downsample(X, n=5, method='decimate')
```

### 8.2 API reference

`sequentia.preprocessing.downsample(X, n, method='decimate')`  
Downsamples an observation sequence (or multiple sequences) by either:

- Decimating the next  $n - 1$  observations
- Averaging the current observation with the next  $n - 1$  observations

#### Parameters

**X:** `numpy.ndarray` or `List[numpy.ndarray]` An individual observation sequence or a list of multiple observation sequences.

**n: int** Downsample factor.

**method: {'decimate', 'average'}** The downsampling method.

**Returns**

**downsampled: numpy.ndarray or List[numpy.ndarray]** The downsampled input observation sequence(s).

---

## Filtering (`filtrate`)

---

Filtering removes or reduces some unwanted components (such as noise) from an observation sequence according to some window size and one of two methods: **median** and **mean** filtering.

Suppose we have an observation sequence  $\mathbf{o}^{(1)}\mathbf{o}^{(2)}\dots\mathbf{o}^{(T)}$  and we are filtering with a window size of  $n$ . Filtering replaces every observation  $\mathbf{o}^{(t)}$  with either the mean or median of the window of observations of size  $n$  containing  $\mathbf{o}^{(t)}$  in its centre.

- For median filtering:  $\mathbf{o}^{(t)'} = \text{med} \left[ \underbrace{\dots, \mathbf{o}^{(t-1)}, \mathbf{o}^{(t)}, \mathbf{o}^{(t+1)}, \dots}_n \right]$
- For mean filtering:  $\mathbf{o}^{(t)'} = \text{mean} \left[ \underbrace{\dots, \mathbf{o}^{(t-1)}, \mathbf{o}^{(t)}, \mathbf{o}^{(t+1)}, \dots}_n \right]$

For further information, please see the [preprocessing tutorial notebook](#).

### 9.1 Example

```

1 import numpy as np
2 from sequentia.preprocessing import filtrate
3
4 # Create some sample data
5 X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
6
7 # Filter the data with window size 5 and median filtering
8 X = filtrate(X, n=5, method='median')

```

### 9.2 API reference

`sequentia.preprocessing.filtrate` ( $X, n, \text{method}='median'$ )  
 Applies a median or mean filter to the input observation sequence(s).

**Parameters**

**X:** `numpy.ndarray` or `List[numpy.ndarray]` An individual observation sequence or a list of multiple observation sequences.

**n:** `int` Window size.

**method:** `{'median', 'mean'}` The filtering method.

**Returns**

**filtered:** `numpy.ndarray` or `List[numpy.ndarray]` The filtered input observation sequence(s).



---

## Discrete Fourier Transform (`fft`)

---

The Discrete Fourier Transform (DFT) converts the observation sequence into a real-valued, same-length sequence of equally-spaced samples of the [discrete-time Fourier transform](#).

The popular [Fast Fourier Transform \(FFT\)](#) implementation is used to efficiently compute the DFT.

For further information, please see the [preprocessing tutorial notebook](#).

### 10.1 Example

```
1 import numpy as np
2 from sequentia.preprocessing import fft
3
4 # Create some sample data
5 X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
6
7 # Transform the data
8 X = fft(X)
```

### 10.2 API reference

`sequentia.preprocessing.fft(X)`

Applies a Discrete Fourier Transform to the input observation sequence(s).

#### Parameters

**X:** `numpy.ndarray` or `List[numpy.ndarray]` An individual observation sequence or a list of multiple observation sequences.

#### Returns

**transformed:** `numpy.ndarray` or `List[numpy.ndarray]` The transformed input observation sequence(s).



---

## Combined Preprocessing (Preprocess)

---

The `Preprocess` class provides a way of efficiently applying multiple preprocessing transformations to provided input observation sequences.

For further information, please see the [preprocessing tutorial notebook](#).

### 11.1 Example

```
1 import numpy as np
2 from sequentia.preprocessing import Preprocess
3
4 # Create some sample data
5 X = [np.random.random((20 * i, 3)) for i in range(1, 4)]
6
7 # Create the Preprocess object
8 pre = Preprocess()
9 pre.trim_zeros()
10 pre.center()
11 pre.standardize()
12 pre.filtrate(n=5, method='median')
13 pre.downsample(n=5, method='decimate')
14 pre.fft()
15
16 # View a summary of the preprocessing steps
17 pre.summary()
18
19 # Transform the data applying transformations in order
20 X = pre.transform(X)
```

## 11.2 API reference

**class** `sequentia.preprocessing.Preprocess`

Efficiently applies multiple preprocessing transformations to the provided input observation sequence(s).

**trim\_zeros** (*self*)

Trim zero-observations from the input observation sequence(s).

**center** (*self*)

Centers an observation sequence (or multiple sequences) by centering observations around the mean.

**standardize** (*self*)

Standardizes an observation sequence (or multiple sequences) by transforming observations so that they have zero mean and unit variance.

**downsample** (*self*, *n*, *method*='decimate')

Downsamples an observation sequence (or multiple sequences) by either:

- Decimating the next  $n - 1$  observations
- Averaging the current observation with the next  $n - 1$  observations

### Parameters

**n: int** Downsample factor.

**method: {'decimate', 'average'}** The downsampling method.

**fft** (*self*)

Applies a Discrete Fourier Transform to the input observation sequence(s).

**filtrate** (*self*, *n*, *method*='median')

Applies a median or mean filter to the input observation sequence(s).

### Parameters

**n: int** Window size.

**method: {'median', 'mean'}** The filtering method.

**transform** (*self*, *X*)

Applies the preprocessing transformations to the provided input observation sequence(s).

### Parameters

**X: numpy.ndarray or List[numpy.ndarray]** An individual observation sequence or a list of multiple observation sequences.

### Returns

**transformed: numpy.ndarray or List[numpy.ndarray]** The input observation sequence(s) with preprocessing transformations applied in order.

**summary** (*self*)

Displays an ordered summary of the preprocessing transformations.

## CHAPTER 12

---

### Documentation Search and Index

---

- search
- genindex



**S**

`sequentia.preprocessing`, 21





## C

`center()` (in module *sequentia.preprocessing*), 21  
`center()` (*sequentia.preprocessing.Preprocess*  
*method*), 32

## D

`downsample()` (in module *sequentia.preprocessing*),  
 25  
`downsample()` (*sequentia.preprocessing.Preprocess*  
*method*), 32  
 DTWKNN (class in *sequentia.classifiers.dtwknn*), 16

## E

`evaluate()` (*sequentia.classifiers.dtwknn.DTWKNN*  
*method*), 16  
`evaluate()` (*sequentia.classifiers.hmm.HMMClassifier*  
*method*),  
 13

## F

`fft()` (in module *sequentia.preprocessing*), 29  
`fft()` (*sequentia.preprocessing.Preprocess*  
*method*), 32  
`filtrate()` (in module *sequentia.preprocessing*), 27  
`filtrate()` (*sequentia.preprocessing.Preprocess*  
*method*), 32  
`fit()` (*sequentia.classifiers.dtwknn.DTWKNN*  
*method*),  
 16  
`fit()` (*sequentia.classifiers.hmm.HMM*  
*method*), 9  
`fit()` (*sequentia.classifiers.hmm.HMMClassifier*  
*method*), 12  
`forward()` (*sequentia.classifiers.hmm.HMM*  
*method*),  
 9

## H

HMM (class in *sequentia.classifiers.hmm*), 9  
 HMMClassifier (class in *sequentia.classifiers.hmm*),  
 12

## P

`predict()` (*sequentia.classifiers.dtwknn.DTWKNN*  
*method*), 16  
`predict()` (*sequentia.classifiers.hmm.HMMClassifier*  
*method*), 12  
 Preprocess (class in *sequentia.preprocessing*), 32

## S

*sequentia.preprocessing* (module), 21  
`set_random_initial()` (*sequentia.classifiers.hmm.HMM*  
*method*), 9  
`set_random_transitions()` (*sequentia.classifiers.hmm.HMM*  
*method*), 9  
`set_uniform_initial()` (*sequentia.classifiers.hmm.HMM*  
*method*), 9  
`set_uniform_transitions()` (*sequentia.classifiers.hmm.HMM*  
*method*), 9  
`standardize()` (in module *sequentia.preprocessing*),  
 23  
`standardize()` (*sequentia.preprocessing.Preprocess*  
*method*), 32  
`summary()` (*sequentia.preprocessing.Preprocess*  
*method*), 32

## T

`transform()` (*sequentia.preprocessing.Preprocess*  
*method*), 32  
`trim_zeros()` (in module *sequentia.preprocessing*),  
 19  
`trim_zeros()` (*sequentia.preprocessing.Preprocess*  
*method*), 32