sequentia

Release 0.7.0

May 19, 2020

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.

CHAPTER 1

Changelog

1.1 0.7.0

1.1.1 Major changes

- Fix `pomegranate` version to v0.12.0. (#79)
- Add serialization and deserialization support for all classifiers. (#80)
  - `HMM`, `HMMClassifier`: Serialized in JSON format.
  - *KNNClassifier*: Serialized in [HDF5](https://support.hdfgroup.org/HDF5/doc/H5.intro.html) format.
- Finish preprocessing documentation and tests. (#81)
- (*Internal*) Remove nested helper functions in `KNNClassifier.predict()`. (#84)
- Add strict left-right HMM topology. (#85)**Note**: This is the more traditional left-right HMM topology.
- Implement GMM-HMMs in the `GMMHMM` class. (#87)
- Implement custom, uniform and frequency-based HMM priors. (#88)
- Implement distance-weighted DTW-kNN predictions. (#90)
- Rename `DTWKNN` to `KNNClassifer`. (#91)

1.1.2 Minor changes

- (*Internal*) Simplify package imports. (#82)
- (*Internal*) Add `Validator.func()` for validating callables. (#90)

1.2 v0.7.0a1

1.2.1 Major changes

- Clean up package imports. (#77)
- Rework `preprocessing` module. (#75)

1.2.2 Minor changes

- Fix typos and update preprocessing information in `README.md`. (#76)

1.3 0.6.1

1.3.1 Major changes

- Remove strict requirement of Numpy arrays being two-dimensional by using `numpy.atleast_2d` to convert one-dimensional arrays into 2D. (#70)

1.3.2 Minor changes

- As the HMM classifier is not a true ensemble of HMMs (since each HMM doesn't really contribute to the classification), it is no longer referred to as an ensemble. (#69)

1.4 0.6.0

1.4.1 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)

1.4.2 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)

1.5 0.5.0

1.5.1 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.

1.5.2 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)

1.6 0.4.0

1.6.1 Major changes

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

1.6.2 Minor changes

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

1.7 0.3.0

1.7.1 Major changes

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

1.8 0.2.0

1.8.1 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.

1.8.2 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)

1.9 0.1.0

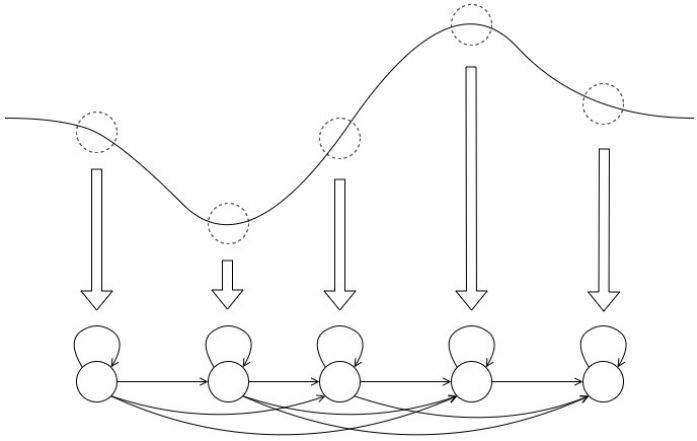
1.9.1 Major changes

Nothing, initial release!

CHAPTER 2

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 π**:
  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 $\mathbf{o}^{(t)} = \left(o_1^{(t)}, o_2^{(t)}, \ldots, o_D^{(t)}\right)$. 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 $\lambda_c = (\boldsymbol{\pi}_c, A_c, B_c)$. We can use the Baum-Welch algorithm (an application of the Expectation-Maximization algorithm) to fit λ_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.

A **strict left-right** topology is one in which transitions are only permitted to the current state and the next state, i.e. no state-jumping is permitted.

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 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 all three topologies, specified by a string parameter `topology` in the `HMM` constructor that takes values *'left-right'*, *'strict-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 λ_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 a group 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 occuring more frequently than others.

2.3 Example

```
import numpy as np
from sequentia.classifiers import HMM

# Create some sample data
X = [np.random.random((10 * i, 3)) for i in range(1, 4)]

# Create and fit a left-right HMM with random transitions and initial state␣
↪distribution
hmm = HMM(label='class1', n_states=5, topology='left-right')
```

(continues on next page)

(continued from previous page)

```
hmm.set_random_initial()
hmm.set_random_transitions()
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', 'strict-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.

as_dict(*self*)
Serializes the *HMM* object into a *dict*, ready to be stored in JSON format.

Returns

serialized: dict JSON-ready serialization of the *HMM* object.

save(*self*, *path*)
Converts the *HMM* object into a *dict* and stores it in a JSON file.

Parameters

path: str File path (with or without *.json* extension) to store the JSON-serialized *HMM* object.

See also:

as_dict Generates the *dict* that is stored in the JSON file.

classmethod load(*data*, *random_state=None*)
Deserializes either a *dict* or JSON serialized *HMM* object.

Parameters

data: str or dict

- File path of the serialized JSON data generated by the *save()* method.
- *dict* representation of the *HMM*, generated by the *as_dict()* method.

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

Returns

deserialized: *HMM* The deserialized HMM object.

See also:

save Serializes a *HMM* into a JSON file.

as_dict Generates a *dict* representation of the *HMM*.

CHAPTER 3

Hidden Markov Model with Gaussian Mixture Emissions (GMMHMM)

The assumption that a single multivariate Gaussian emission distribution is accurate and representative enough to model the probability of observation vectors of any state of a HMM is often a very strong and naive one.

Instead, a more powerful approach is to represent the emission distribution as a mixture of multiple multivariate Gaussian densities. An emission distribution for state m, formed by a mixture of G multivariate Gaussian densities is defined as:

$$b_m(\mathbf{o}^{(t)}) = \sum_{g=1}^{G} c_g^{(m)} \mathcal{N}\left(\mathbf{o}^{(t)} \,;\, \boldsymbol{\mu}_g^{(m)}, \Sigma_g^{(m)}\right)$$

where $\mathbf{o}^{(t)}$ is an observation vector at time t, $c_g^{(m)}$ is a *mixing coefficient* such that $\sum_{g=1}^{G} c_g^{(m)} = 1$ and $\boldsymbol{\mu}_g^{(m)}$ and $\Sigma_g^{(m)}$ are the mean vector and covariance matrix of the g^{th} mixture component of the m^{th} state, respectively.

Even in the case that multiple Gaussian densities are not needed, the mixing coefficients can be adjusted so that irrelevant Gaussians are omitted and only a single Gaussian remains.

3.1 Example

```
import numpy as np
from sequentia.classifiers import GMMHMM

# Create some sample data
X = [np.random.random((10 * i, 3)) for i in range(1, 4)]

# Create and fit a left-right HMM with random transitions and initial state distribution
hmm = GMMHMM(label='class1', n_states=5, n_components=3, covariance='diagonal', topology='left-right')
hmm.set_random_initial()
hmm.set_random_transitions()
hmm.fit(X)
```

3.2 API reference

class sequentia.classifiers.hmm.**GMMHMM**(*label*, *n_states*, *n_components*, *covariance='diagonal'*, *topology='left-right'*, *random_state=None*)

A hidden Markov model representing an isolated temporal sequence class, with mixtures of multivariate Gaussian components representing state emission distributions.

Parameters

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

n_states: int The number of states for the model.

n_components: int The number of mixture components used in the emission distribution for each state.

covariance: {'diagonal', 'full'} The covariance matrix type.

topology: {'ergodic', 'left-right', 'strict-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.

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.

as_dict(*self*)

Serializes the *GMMHMM* object into a *dict*, ready to be stored in JSON format.

Returns

serialized: dict JSON-ready serialization of the *GMMHMM* object.

classmethod load(*data*, *random_state=None*)

Deserializes either a *dict* or JSON serialized *GMMHMM* object.

Parameters

data: str or dict

- File path of the serialized JSON data generated by the *save()* method.
- *dict* representation of the *GMMHMM*, generated by the *as_dict()* method.

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

Returns

deserialized: *GMMHMM* The deserialized HMM object.

See also:

save Serializes a *GMMHMM* into a JSON file.

as_dict Generates a *dict* representation of the *GMMHMM*.

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.

save(*self*, *path*)

Converts the *HMM* object into a *dict* and stores it in a JSON file.

Parameters

path: str File path (with or without *.json* extension) to store the JSON-serialized *HMM* object.

See also:

as_dict Generates the *dict* that is stored in the JSON file.

set_random_initial(*self*)

Sets a random initial state distribution.

set_random_transitions(*self*)

Sets a random transition matrix according to the topology.

set_uniform_initial(*self*)

Sets a uniform initial state distribution.

set_uniform_transitions(*self*)

Sets a uniform transition matrix according to the topology.

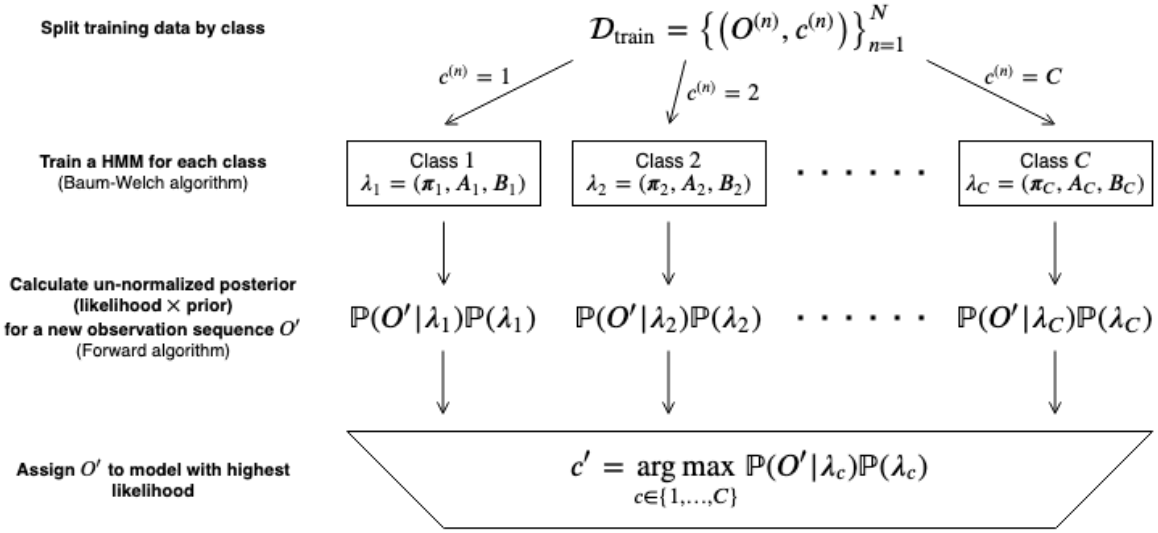
CHAPTER 4

Hidden Markov Model Classifier (`HMMClassifier`)

Multiple HMMs (and/or GMM-HMMs) can be combined to form a multi-class classifier. To classify a new observation sequence O', this works by:

1. Creating and training the HMMs $\lambda_1, \lambda_2, \ldots, \lambda_C$.
2. Calculating the likelihoods $\mathbb{P}(O'|\lambda_1), \mathbb{P}(O'|\lambda_2), \ldots, \mathbb{P}(O'|\lambda_C)$ of each model generating O'.
3. Scaling the likelihoods by priors $\mathbb{P}(\lambda_1), \mathbb{P}(\lambda_2), \ldots, \mathbb{P}(\lambda_C)$, producing un-normalized posteriors $\mathbb{P}(O'|\lambda_c)\mathbb{P}(\lambda_c)$.
4. Performing MAP classification by choosing the class represented by the HMM with the highest posterior – that is, $c^* = \arg\max_{c \in \{1,\ldots,C\}} \mathbb{P}(O'|\lambda_c)\mathbb{P}(\lambda_c)$.

These steps are summarized in the diagram below.



4.1 Example

```
import numpy as np
from sequentia.classifiers import HMM, HMMClassifier

# Set of possible labels
labels = ['class{}'.format(i) for i in range(5)]

# Create and fit some sample HMMs
hmms = []
for i, label in enumerate(labels):
    hmm = HMM(label=label, n_states=(i + 3), topology='left-right')
    hmm.set_random_initial()
    hmm.set_random_transitions()
    hmm.fit([np.arange((i + j * 20) * 30).reshape(-1, 3) for j in range(1, 4)])
    hmms.append(hmm)

# Create some sample test data and labels
X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
y = ['class0', 'class1', 'class1']

# Create a classifier and calculate predictions and evaluations
clf = HMMClassifier()
clf.fit(hmms)
predictions = clf.predict(X)
accuracy, confusion = clf.evaluate(X, y, labels=labels)
```

For more elaborate examples, please have a look at the example notebooks.

4.2 API reference

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

A classifier that combines individual *HMM* and/or *GMMHMM* objects, which model isolated sequences from different classes.

fit (*self*, *models*)

Fits the classifier with a collection of *HMM* and/or *GMMHMM* objects.

Parameters

models: List[HMM, GMMHMM] or Dict[Any, HMM/GMMHMM] A collection of *HMM* objects to use for classification.

predict (*self*, *X*, *prior='frequency'*, *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: {'frequency', 'uniform'} or Dict[str, float] How the prior for each model is calculated to perform MAP estimation by scoring with the joint probability (or un-normalized posterior) $\mathbb{P}(O, \lambda_c) = \mathbb{P}(O|\lambda_c)\mathbb{P}(\lambda_c)$, where the likelihood $\mathbb{P}(O|\lambda_c)$ is generated from the models' *forward()* function.

- *'frequency'*: Calculate the prior $\mathbb{P}(\lambda_c)$ as the proportion of training examples in class c.
- *'uniform'*: Set the priors uniformly such that $\mathbb{P}(\lambda_c) = \frac{1}{C}$ for each class $c \in \{1, \ldots, C\}$.

Alternatively, class priors can be specified in a `dict`, e.g. `{'class1': 0.1, 'class2': 0.3, 'class3': 0.6}`.

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='frequency'*, *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: {'frequency', 'uniform'} or Dict[str, float] How the prior for each model is calculated to perform MAP estimation by scoring with the joint probability (or un-normalized posterior) $\mathbb{P}(O, \lambda_c) = \mathbb{P}(O|\lambda_c)\mathbb{P}(\lambda_c)$, where the likelihood $\mathbb{P}(O|\lambda_c)$ is generated from the models' *forward()* function.

- *'frequency'*: Calculate the prior $\mathbb{P}(\lambda_c)$ as the proportion of training examples in class c.
- *'uniform'*: Set the priors uniformly such that $\mathbb{P}(\lambda_c) = \frac{1}{C}$ for each class $c \in \{1, \ldots, C\}$.

Alternatively, class priors can be specified in a `dict`, e.g. `{'class1': 0.1, 'class2': 0.3, 'class3': 0.6}`.

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.

as_dict(*self*)

Serializes the *HMMClassifier* object into a *dict*, ready to be stored in JSON format.

Note: Serializing a *HMMClassifier* implicitly serializes the internal *HMM* or *GMMHMM* objects by calling *HMM.as_dict()* or *GMMHMM.as_dict()* and storing all of the model data in a single *dict*.

Returns

serialized: dict JSON-ready serialization of the *HMMClassifier* object.

See also:

HMM.as_dict The serialization function used for individual *HMM* objects.

GMMHMM.as_dict The serialization function used for individual *GMMHMM* objects.

save(*self*, *path*)
Converts the *HMMClassifier* object into a *dict* and stores it in a JSON file.

Parameters

path: str File path (with or without *.json* extension) to store the JSON-serialized *HMMClassifier* object.

See also:

as_dict Generates the *dict* that is stored in the JSON file.

classmethod load(*path*, *random_state=None*)
Deserializes either a *dict* or JSON serialized *HMMClassifier* object.

Parameters

path: str File path of the serialized JSON data generated by the *save()* method.

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

Returns

deserialized: *HMMClassifier* The deserialized HMM classifier object.

See also:

save Serializes a *HMMClassifier* into a JSON file.

as_dict Generates a *dict* representation of the *HMMClassifier*.

CHAPTER 5

Dynamic Time Warping *k*-Nearest Neighbors Classifier (`KNNClassifier`)

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 `KNNClassifier` class.

5.1 Example

```
import numpy as np
from sequentia.classifiers import KNNClassifier

# Create some sample data
X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
y = ['class0', 'class1', 'class1']
```

(continues on next page)

(continued from previous page)

```
7
8  # Create and fit the classifier
9  clf = KNNClassifier(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.

5.2 API reference

class sequentia.classifiers.knn.**KNNClassifier**(*k*, *radius*, *metric=<function euclidean>*, *weighting=<function KNNClassifier.<lambda>>*)

A k-Nearest Neighbor classifier that compares differing length observation sequences using the efficient FastDTW 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.

weighting: callable A function that specifies how distance weighting should be performed. Using a constant-valued function to set all weights equally is equivalent to no weighting (which is the default configuration). Common weighting functions are $e^{-\alpha x}$ or $\frac{1}{x}$, where x is the DTW distance between two observation sequences.

A weighting function should *ideally* be defined at $x = 0$ in the rare event that two observation sequences are perfectly aligned (i.e. have zero DTW distance).

- **Input**: $x \geq 0$, a DTW distance between two observation sequences.
- **Output**: A floating point value representing the weight used to perform nearest neighbor classification.

Note: Depending on your distance *metric*, it may be desirable to restrict DTW distances to a small range if you intend to use a weighting function.

Using the `MinMaxScale` or `Standardize` preprocessing transformations to scale your features helps to ensure that distances remain small.

---

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.

save(*self*, *path*)
Stores the *KNNClassifier* object into a HDF5 file.

Parameters

path: str File path (with or without *.h5* extension) to store the HDF5-serialized *KNNClassifier* object.

classmethod load(*path*, *encoding='utf-8'*, *metric=<function euclidean at 0x7febdbe82158>*, *weighting=<function KNNClassifier.<lambda> at 0x7febce45a2f0>*)
Deserializes a HDF5-serialized *KNNClassifier* object.

Parameters

path: str File path of the serialized HDF5 data generated by the *save()* method.

encoding: str The encoding used to represent training labels when decoding the HDF5 file.

Note: Supported string encodings in Python can be found here.

---

metric: callable Distance metric for FastDTW (see *KNNClassifier*).

weighting: callable A function that specifies how distance weighting should be performed (see *KNNClassifier*).

Returns

deserialized: *KNNClassifier* The deserialized DTW k-NN classifier object.

See also:

save Serializes a *KNNClassifier* into a HDF5 file.

CHAPTER 6

Introduction to Preprocessing

Sequentia provides a number of useful preprocessing methods for sequential data.

- *Length Equalizing* (`Equalize`)
- *Zero Trimming* (`TrimZeros`)
- *Min-max Scaling* (`MinMaxScale`)
- *Centering* (`Center`)
- *Standardizing* (`Standardize`)
- *Downsampling* (`Downsample`)
- *Filtering* (`Filter`)

Additionally, the provided `Preprocess` class makes it possible to *apply multiple transformations*.

Each of the transformations follow a similar interface, based on the abstract `Transform` class:

class sequentia.preprocessing.**Transform**

Base class representing a single transformation.

transform(*self*, *X*, *verbose=False*)

Applies the transformation.

Parameters

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

verbose: bool Whether or not to display a progress bar when applying transformations.

Returns

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

__call__(*self*, *X*, *verbose=False*)

Alias of the *transform()* method.

fit (*self*, *X*)
Fit the transformation on the provided observation sequence(s) (without transforming them).

Parameters

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

fit_transform (*self*, *X*, *verbose=False*)
Fit the transformation with the provided observation sequence(s) and transform them.

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting and applying transformations.

Returns

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

CHAPTER 7

Length Equalizing (`Equalize`)

7.1 API reference

class sequentia.preprocessing.**Equalize**

Equalize all observation sequence lengths by padding or trimming zeros.

fit(*self*, *X*)

Fit the transformation on the provided observation sequence(s) (without transforming them).

Parameters

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

transform(*self*, *X*, *verbose=False*)

Applies the transformation.

Parameters

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

verbose: bool Whether or not to display a progress bar when applying transformations.

Returns

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

__call__(*self*, *X*, *verbose=False*)

Alias of the *transform()* method.

fit_transform(*self*, *X*, *verbose=False*)

Fit the transformation with the provided observation sequence(s) and transform them.

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting and applying transformations.

Returns

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

CHAPTER 8

Zero Trimming (`TrimZeros`)

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 Sequentia 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.

8.1 API reference

class sequentia.preprocessing.**TrimZeros**

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

Examples

```
>>> # Create some sample data
>>> z = np.zeros((4, 3))
>>> x = lambda i: np.vstack((z, np.random.random((10 * i, 3)), z))
>>> X = [x(i) for i in range(1, 4)]
>>> # Zero-trim the data
>>> X = TrimZeros()(X)
```

transform(*self*, *X*, *verbose=False*)

Applies the transformation.

Parameters

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

verbose: bool Whether or not to display a progress bar when applying transformations.

Returns

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

__call__(*self*, *X*, *verbose=False*)
Alias of the *transform()* method.

fit(*self*, *X*)
Fit the transformation on the provided observation sequence(s) (without transforming them).

Parameters

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

fit_transform(*self*, *X*, *verbose=False*)
Fit the transformation with the provided observation sequence(s) and transform them.

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting and applying transformations.

Returns

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

CHAPTER 9

Min-max Scaling (`MinMaxScale`)

9.1 API reference

class sequentia.preprocessing.**MinMaxScale**(*scale=(0, 1)*, *independent=True*)

Scales the observation sequence features to each be within a provided range.

Parameters

scale: tuple(int, int) The range of the transformed observation sequence features.

independent: bool Whether to independently compute the minimum and maximum to scale each observation sequence.

fit(*self*, *X*)

Fit the transformation on the provided observation sequence(s) (without transforming them).

Parameters

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

transform(*self*, *X*, *verbose=False*)

Applies the transformation.

Parameters

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

verbose: bool Whether or not to display a progress bar when applying transformations.

Returns

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

__call__(*self*, *X*, *verbose=False*)

Alias of the *transform()* method.

fit_transform(*self*, *X*, *verbose=False*)

Fit the transformation with the provided observation sequence(s) and transform them.

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting and applying transformations.

Returns

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

CHAPTER 10

Centering (`Center`)

10.1 API reference

class sequentia.preprocessing.**Center**(*independent=True*)

Centers the observation sequence features around their means. Results in zero-mean features.

Parameters

independent: bool Whether to independently compute the mean to scale each observation sequence.

Examples

```
>>> # Create some sample data
>>> X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
>>> # Center the data
>>> X = Center()(X)
```

fit(*self*, *X*)

Fit the transformation on the provided observation sequence(s) (without transforming them).

Parameters

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

transform(*self*, *X*, *verbose=False*)

Applies the transformation.

Parameters

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

verbose: bool Whether or not to display a progress bar when applying transformations.

Returns

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

__call__(*self*, *X*, *verbose=False*)
Alias of the *transform()* method.

fit_transform(*self*, *X*, *verbose=False*)
Fit the transformation with the provided observation sequence(s) and transform them.

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting and applying transformations.

Returns

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

CHAPTER 11

Standardizing (`Standardize`)

11.1 API reference

class sequentia.preprocessing.**Standardize**(*independent=True*)

Centers the observation sequence features around their means, then scales them by their deviations. Results in zero-mean, unit-variance features.

Parameters

independent: bool Whether to independently compute the mean and standard deviation to scale each observation sequence.

Examples

```
>>> # Create some sample data
>>> X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
>>> # Standardize the data
>>> X = Standardize()(X)
```

fit(*self*, *X*)

Fit the transformation on the provided observation sequence(s) (without transforming them).

Parameters

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

transform(*self*, *X*, *verbose=False*)

Applies the transformation.

Parameters

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

verbose: bool Whether or not to display a progress bar when applying transformations.

Returns

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

__call__(*self*, *X*, *verbose=False*)
Alias of the *transform()* method.

fit_transform(*self*, *X*, *verbose=False*)
Fit the transformation with the provided observation sequence(s) and transform them.

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting and applying transformations.

Returns

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

CHAPTER 12

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.

12.1 API reference

class `sequentia.preprocessing.`**Downsample**(*factor*, *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

factor: int > 0 Downsample factor.

method: {'decimate', 'mean'} The downsampling method.

Examples

```
>>> # Create some sample data
>>> X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
>>> # Downsample the data with downsample factor 5 and decimation
>>> X = Downsample(factor=5, method='decimate')(X)
```

transform(*self*, *X*, *verbose=False*)

Applies the transformation.

Parameters

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

verbose: bool Whether or not to display a progress bar when applying transformations.

Returns

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

__call__(*self*, *X*, *verbose=False*)
Alias of the `transform()` method.

fit(*self*, *X*)
Fit the transformation on the provided observation sequence(s) (without transforming them).

Parameters

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

fit_transform(*self*, *X*, *verbose=False*)
Fit the transformation with the provided observation sequence(s) and transform them.

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting and applying transformations.

Returns

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

CHAPTER 13

Filtering (`Filter`)

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: $\mathbf{o}^{(t)\prime} = \text{med}\underbrace{\left[\dots,\mathbf{o}^{(t-1)},\mathbf{o}^{(t)},\mathbf{o}^{(t+1)},\dots\right]}_{n}$
- For mean filtering: $\mathbf{o}^{(t)\prime} = \text{mean}\underbrace{\left[\dots,\mathbf{o}^{(t-1)},\mathbf{o}^{(t)},\mathbf{o}^{(t+1)},\dots\right]}_{n}$

13.1 API reference

class sequentia.preprocessing.**Filter**(*window_size*, *method='median'*)

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

Parameters

window_size: int The size of the filtering window.

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

Examples

```
>>> # Create some sample data
>>> X = [np.random.random((10 * i, 3)) for i in range(1, 4)]
>>> # Filter the data with window size 5 and median filtering
>>> X = Filter(window_size=5, method='median')(X)
```

transform(*self*, *X*, *verbose=False*)
Applies the transformation.

Parameters

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

verbose: bool Whether or not to display a progress bar when applying transformations.

Returns

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

__call__(*self*, *X*, *verbose=False*)
Alias of the *transform()* method.

fit(*self*, *X*)
Fit the transformation on the provided observation sequence(s) (without transforming them).

Parameters

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

fit_transform(*self*, *X*, *verbose=False*)
Fit the transformation with the provided observation sequence(s) and transform them.

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting and applying transformations.

Returns

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

CHAPTER 14

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.

14.1 API reference

class sequentia.preprocessing.**Preprocess**(*steps*)

A pipeline of preprocessing transformations.

Parameters

steps: List[Transform] A list of preprocessing transformations.

Examples

```
>>> # Create some sample data
>>> X = [np.random.random((20 * i, 3)) for i in range(1, 4)]
>>> # Create the Preprocess object
>>> pre = Preprocess([
>>>     TrimZeros(),
>>>     Center(),
>>>     Standardize(),
>>>     Filter(window_size=5, method='median'),
>>>     Downsample(factor=5, method='decimate')
>>> ])
>>> # View a summary of the preprocessing steps
>>> pre.summary()
>>> # Transform the data applying transformations in order
>>> X = pre(X)
```

transform(*self*, *X*, *verbose=False*)

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.

verbose: bool Whether or not to display a progress bar when applying transformations.

Returns

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

fit(*self*, *X*, *verbose=False*)

Fit the preprocessing transformations with the provided observation sequence(s).

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting transformations.

fit_transform(*self*, *X*, *verbose=False*)

Fit the preprocessing transformations with the provided observation sequence(s) and transform them.

Parameters

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

verbose: bool Whether or not to display a progress bar when fitting and applying transformations.

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 15

Documentation Search and Index

- search
- genindex

Index

Symbols

A

C

D

E

F

G

H

K

L

M

P

S

T