sequentia

Release 0.5.0

Jan 01, 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 0.5.0

1.1.1 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. (#53)
- Modify preprocessing boundary conditions (#51):
  - Use a bi-directional window for filtering to resolve boundary problems.
  - Modify downsampling method to downsample residual observations.

1.1.2 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)

1.2 0.4.0

1.2.1 Major changes

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

1.2.2 Minor changes

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

1.3 0.3.0

1.3.1 Major changes

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

1.4 0.2.0

1.4.1 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)
- 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.

1.4.2 Minor changes

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

1.5 0.1.0

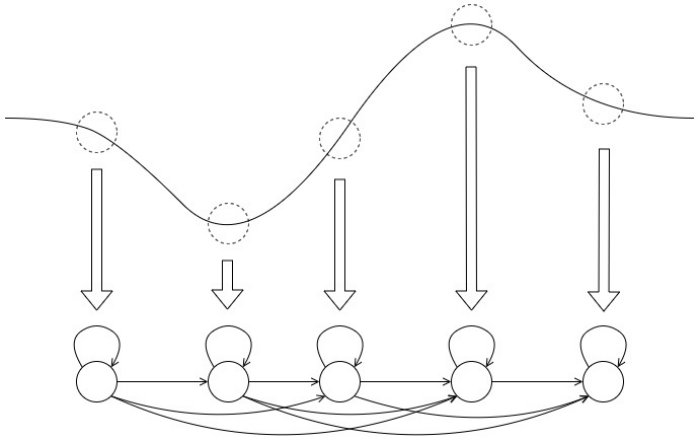
1.5.1 Major changes

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.

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 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 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 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')
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'} 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.

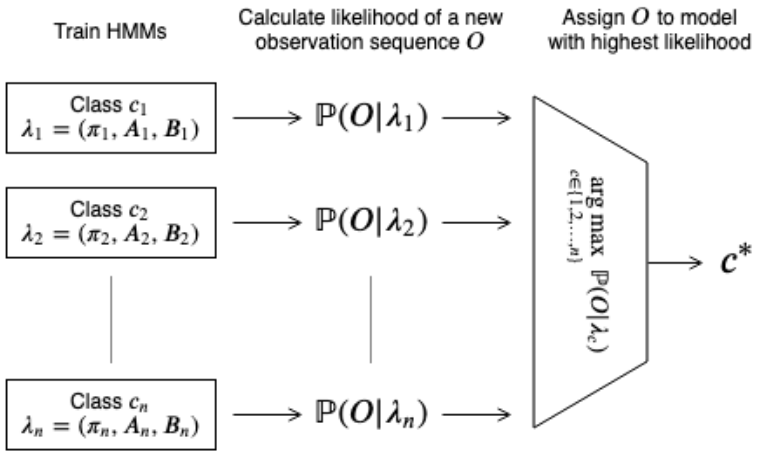
CHAPTER 3

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 $\lambda_1, \lambda_2, \ldots, \lambda_N$.
2. Calculating the likelihoods $\mathbb{P}(O'|\lambda_1), \mathbb{P}(O'|\lambda_2), \ldots, \mathbb{P}(O'|\lambda_N)$ of each model generating O'.
   Note: You can also used 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,
   $c^* = \arg\max_{c \in \{1,\ldots,N\}} \mathbb{P}(O'|\lambda_c)$.

These steps are summarized in the diagram below.



3.1 Example

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

# Set of possible labels
labels = [f'class{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.

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.

CHAPTER 4

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

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

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

# Create and fit the classifier
```

(continues on next page)

(continued from previous page)

```
clf = DTWKNN(k=1, radius=5)
clf.fit(X, y)

# Predict labels for the training data (just as an example)
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 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.

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.

CHAPTER 5

Normalization (`normalize`)

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

$$O = \begin{pmatrix} o_1^{(1)} & o_2^{(1)} & \cdots & o_D^{(1)} \\ o_1^{(2)} & o_2^{(2)} & \cdots & o_D^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ o_1^{(T)} & o_2^{(T)} & \cdots & o_D^{(T)} \end{pmatrix} \qquad \boldsymbol{\mu} = \begin{pmatrix} \overline{o_1} & \overline{o_2} & \cdots & \overline{o_D} \end{pmatrix}$$

Where $\overline{o_d}$ represents the mean of the d^{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.

5.1 Example

```
import numpy as np
from sequentia.preprocessing import normalize

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

# Normalize the data
X = normalize(X)
```

5.2 API reference

`sequentia.preprocessing.`**`normalize`**`(X)`

Normalizes 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

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

CHAPTER 6

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.

6.1 Example

```
import numpy as np
from sequentia.preprocessing import downsample

# 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(X, n=5, method='decimate')
```

6.2 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).

CHAPTER 7

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

For further information, please see the preprocessing tutorial notebook.

7.1 Example

```
import numpy as np
from sequentia.preprocessing import filtrate

# 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 = filtrate(X, n=5, method='median')
```

7.2 API reference

`sequentia.preprocessing.`**`filtrate`**(*X*, *n*, *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).

CHAPTER 8

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.

8.1 Example

```
import numpy as np
from sequentia.preprocessing import fft

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

# Transform the data
X = fft(X)
```

8.2 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).

CHAPTER 9

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.

9.1 Example

```
import numpy as np
from sequentia.preprocessing import Preprocess

# Create some sample data
X = [np.random.random((20 * i, 3)) for i in range(1, 4)]

# Create the Preprocess object
pre = Preprocess()
pre.normalize()
pre.filtrate(n=5, method='median')
pre.downsample(n=5, method='decimate')
pre.fft()

# View a summary of the preprocessing steps
pre.summary()

# Transform the data applying transformations in order
X = pre.transform(X)
```

9.2 API reference

class sequentia.preprocessing.**Preprocess**

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

normalize(*self*)
Normalizes an observation sequence (or multiple sequences) by centering observations around the mean.

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 10

Documentation Search and Index

- search
- genindex
- modindex

Python Module Index

S

Index