Hidden Markov Model Documentation

Release 0.3

Rahul Ramesh

Aug 22, 2017

Table of Contents

Contents:

CHAPTER 1

Introduction

This package is an implementation of Viterbi Algorithm, Forward algorithm and the Baum Welch Algorithm. The computations are done via matrices to improve the algorithm runtime. Package **hidden_markov** is tested with Python version 2.7 and Python version 3.5.

Installation

To install this package, clone this repo and from the root directory run:

```
$ python setup.py install
```

An alternative way to install the package **hidden_markov**, is to use pip or easy_install, i.e. run the command:

```
$ pip install hidden_markov
```

Unfamiliar with pip? Checkout this link to install pip.

Requirements

- numpy

CHAPTER 2

Tutorial

Introduction

A Hidden Markov model is a Markov chain for which the states are not explicitly observable .We instead make indirect observations about the state by events which result from those hidden states .Since these observables are not sufficient/complete to describe the state, we associate a probability with each of the observable coming from a particular state . In other words, if the probability falls to either 1 or 0,it reduces to a Markov model.

1st order Markov assumption :The probability of occurrence of an event at time 't' depends only on the observation at time 't-1' and not on the events that happened before 't-1'. In other words, the observations O1,O2,...Ot-1 do not impact the observation Ot. Hidden Markov models work on this assumption.The initial probability array, transition array and observation/emission array can completely define a HMM.

Basic Definitions

- *O : { O1, O2 , ... OT }* : Sequence of observations
- *Q : {Q1 , Q2 , ... QT }* : Sequence of corresponding hidden states
- *S : {S1 , S2, ... , Sn }* : Set of unique states in the Hidden Markov model
- *M : {M1 , M2, ... , Mn }* : Set of unique observations in the Hidden Markov model
- *n* : Number of hidden states in the model
- *m* : Number of unique observations in the model
- *T* : Transition matrix
- *E* : Emission matrix
- *Pi*: Initial/ Start probability
- *Alpha* : Forward probability values
- *Beta* : backward probability values

Forward/Backward Probability

Given an observation sequence O={O1 , O2,.... OT}, the forward probability Alpha[i,t] is the probability for the sequence O0, O1,...,Ot to end in the state S[i] after 't' stages or at time 't'.

Alpha(i) = P(Qt= Si , O1, O2,...,OT)

Given an observation sequence O={O0 , O1,.... OT}, the backward probability Beta(i) for a state Siat time t is the probability for the sequence Ot+1, Ot+2,...,Ot is observed given the state is Si at time 't'.

Beta(i) = P(Ot+1, Ot+2,...,OT | Qt= Si)

Forward Algorithm

The forward algorithm gives us the probability of an observed sequence in the HMM. So why do we need to find the probability of an observed sequence? This type of problem occurs in speech recognition where a large number of Markov models are used, and each one modelling a particular word. Each utterance of a word, will now give us a set of observation variables.

Hence we will use the Markov model that has the highest probability of this observation sequence. The Forward algorithm is also an important sub-routine of the forward-backward algorithm. The algorithm works by calculating Alpha over various 't'. The sum of alpha for all states for time = 'T'

Viterbi Algorithm

Viterbi algorithm is used to find out the most likely sequence of hidden states that can generate the given set of observations. For example, in speech recognition, the acoustic signal is treated as the observed sequence of events, and a string of text is considered to be the "hidden cause" of the acoustic signal. The Viterbi algorithm finds the most likely string of text given the acoustic signal.

So, what we essentially do is in each step of algorithm, we calculate the probabilities of landing up in another state from any present state. We compare transition probabilities between states. We choose whichever is higher and move on.

Forward-Backward Algorithm

The third and probably the most important of the three algorithms is the forward backward algorithm. Often, the emission, transition and start probabilities are not known. This algorithms aims to find the best model parameters based on the principles of expectation maximization. The algorithm works iteratively and in each iteration, the model parameters are updated, such that the probability of occurrence of a set of observations increase.

See Also

Check this link to view the more detailed tutorial.

CHAPTER 3

Usage

Initialization

class hidden_markov.**hmm**(*states*, *observations*, *start_prob*, *trans_prob*, *em_prob*)

Stores a hidden markov model object, and the model parameters.

Implemented Algorithms :

•Viterbi Algorithm

•Forward Algorithm

•Baum-Welch Algorithm

Initialize The hmm class object.

Arguments:

Parameters

- **states** (*A list or tuple*) – The set of hidden states
- **observations** (*A list or tuple*) – The set unique of possible observations
- **start_prob** (*Numpy matrix, dimension = [length(states) X 1]*) – The start probabilities of various states, given in same order as 'states' variable. **start_prob[i] = probability(start at states[i])**.
- **trans_prob** (*Numpy matrix, dimension = [len(states) X len(states)]*) – The transition probabilities, with ordering same as 'states' variable . **trans_prob[i,j] = probability(states[i] -> states[j])**.
- **em_prob** (*Numpy matrix, dimension = [len(states) X len(observations)]*) – The emission probabilities, with ordering same as 'states' variable and 'observations' variable. **em_prob[i,j] = probability(states[i],observations[j])**.

Example:

```
>>> states = ('s', 't')
>>> possible_observation = ('A','B' )
>>> # Numpy arrays of the data
>>> start_probability = np.matrix( '0.5 0.5 ')
>>> transition_probability = np.matrix('0.6 0.4 ;  0.3 0.7 ')
>>> emission_probability = np.matrix( '0.3 0.7 ; 0.4 0.6 ' )
>>> test = hmm(states,possible_observation,start_probability,transition_
↪probability,emission_probability)
```

Viterbi Algorithm

hmm.**viterbi**(*observations*)

The probability of occurence of the observation sequence

Arguments:

Parameters **observations** (*A list or tuple*) – The observation sequence, where each element belongs to 'observations' variable declared with __init__ object.

Returns Returns a list of hidden states.

Return type list of states

Features:

Scaling applied here. This ensures that no underflow error occurs.

Example:

```
>>> states = ('s', 't')
>>> possible_observation = ('A','B' )
>>> # Numpy arrays of the data
>>> start_probability = np.matrix( '0.5 0.5 ')
>>> transition_probability = np.matrix('0.6 0.4 ;  0.3 0.7 ')
>>> emission_probability = np.matrix( '0.3 0.7 ; 0.4 0.6 ' )
>>> # Initialize class object
>>> test = hmm(states,possible_observation,start_probability,transition_
↪probability,emission_probability)
>>> observations = ('A', 'B','B','A')
>>> print(test.viterbi(observations))
```

Forward Algorithm

hmm.**forward_algo**(*observations*)

Finds the probability of an observation sequence for given model parameters

Arguments:

Parameters **observations** (*A list or tuple*) – The observation sequence, where each element belongs to 'observations' variable declared with __init__ object.

Returns The probability of occurence of the observation sequence

Return type float

Example:

```
>>> states = ('s', 't')
>>> possible_observation = ('A','B' )
>>> # Numpy arrays of the data
>>> start_probability = np.matrix( '0.5 0.5 ')
>>> transition_probability = np.matrix('0.6 0.4 ;  0.3 0.7 ')
>>> emission_probability = np.matrix( '0.3 0.7 ; 0.4 0.6 ' )
>>> # Initialize class object
>>> test = hmm(states,possible_observation,start_probability,transition_
↪probability,emission_probability)
>>> observations = ('A', 'B','B','A')
>>> print(test.forward_algo(observations))
```

Note: No scaling applied here and hence this routine is susceptible to underflow errors. Use *hmm.log_prob()* instead.

Baum-Welch Algorithm

hmm.**train_hmm**(*observation_list*, *iterations*, *quantities*)

Runs the Baum Welch Algorithm and finds the new model parameters

Arguments:

Parameters

- **observation_list** (*Contains a list multiple observation sequences.*) – A nested list, or a list of lists
- **iterations** (*An integer*) – Maximum number of iterations for the algorithm
- **quantities** (*A list of integers*) – Number of times, each corresponding item in 'observation_list' occurs.

Returns Returns the emission, transition and start probabilites as numpy matrices

Return type Three numpy matices

Features:

Scaling applied here. This ensures that no underflow error occurs.

Example:

```
>>> states = ('s', 't')
>>> possible_observation = ('A','B' )
>>> # Numpy arrays of the data
>>> start_probability = np.matrix( '0.5 0.5 ')
>>> transition_probability = np.matrix('0.6 0.4 ;  0.3 0.7 ')
>>> emission_probability = np.matrix( '0.3 0.7 ; 0.4 0.6 ' )
>>> # Initialize class object
>>> test = hmm(states,possible_observation,start_probability,transition_
↪probability,emission_probability)
>>>
>>> observations = ('A', 'B','B','A')
>>> obs4 = ('B', 'A','B')
>>> observation_tuple = []
>>> observation_tuple.extend( [observations,obs4] )
>>> quantities_observations = [10, 20]
```

```
>>> num_iter=1000
>>> e,t,s = test.train_hmm(observation_tuple,num_iter,quantities_observations)
>>> # e,t,s contain new emission transition and start probabilities
```

Log-Probability Forward algorithm

hmm.**log_prob**(*observations_list*, *quantities*)

Finds Weighted log probability of a list of observation sequences

Arguments:

Parameters

- **observation_list** (*Contains a list multiple observation sequences.*) – A nested list, or a list of lists
- **quantities** (*A list of integers*) – Number of times, each corresponding item in 'observation_list' occurs.

Returns Weighted log probability of multiple observations.

Return type float

Features:

Scaling applied here. This ensures that no underflow error occurs.

Example:

```
>>> states = ('s', 't')
>>> possible_observation = ('A','B' )
>>> # Numpy arrays of the data
>>> start_probability = np.matrix( '0.5 0.5 ')
>>> transition_probability = np.matrix('0.6 0.4 ;  0.3 0.7 ')
>>> emission_probability = np.matrix( '0.3 0.7 ; 0.4 0.6 ' )
>>> # Initialize class object
>>> test = hmm(states,possible_observation,start_probability,transition_
↪probability,emission_probability)
>>> observations = ('A', 'B','B','A')
>>> obs4 = ('B', 'A','B')
>>> observation_tuple = []
>>> observation_tuple.extend( [observations,obs4] )
>>> quantities_observations = [10, 20]
>>>
>>> prob = test.log_prob(observation_tuple, quantities_observations)
```

CHAPTER 4

Example

This notebook illustrates the usage of the functions in this package, for a discrete hidden markov model. In the example below, the HMM has two states 's' and 't'. There are two possible observation which are 'A' and 'B'. The start probabilities, emission probabilities and transition probabilities are initialized as below. There are two observation sequences 'obs1' and 'obs2'. The variable 'quantities_observations' indicates the number of times, the sequences obs1 and obs2 are seen.

```
# Import required Libraries
import numpy as np
from hidden_markov import hmm
```

Parameter Intialization

```
# ====Initializing Parameters ====

# States
states = ('s', 't')

# list of possible observations
possible_observation = ('A','B' )

# The observations that we observe and feed to the model
obs1 = ('A', 'B','B','A')
obs2 = ('B', 'A','B')

# Number of observation sequece 1 and observation sequence 2
quantities_observations = [10, 20]

observation_tuple = []
observation_tuple.extend( [obs1,obs2] )

# Input parameters as Numpy matrices
start_probability = np.matrix( '0.5 0.5 ')
```

```
transition_probability = np.matrix('0.6 0.4 ;  0.3 0.7 ')
emission_probability = np.matrix( '0.3 0.7 ; 0.4 0.6 ' )
```

A class object called 'test' is initialized with parameters. Note that the parameters are mandatory and default arguments do not exist. Additionally, the start probabilities, transition probabilities and emission probabilites are all numpy matrices. The observations and states are both tuples and the observation_tuples variable is a list of observations.

```
test = hmm(states,possible_observation,start_probability,transition_probability,
↪emission_probability)
```

Forward Algorithm

The forward algorithm find the probability of occurence of an observation sequence. The function inputs an observation sequence and returns the probability. The transition, start and emission probabilities used, are the same as those specified in the class object definition.

```
#Forward Algorithm Results on 'obs1'
test.forward_algo(obs1)
```

```
0.051533999999999996
```

Viterbi Algorithm

The Viterbi algorithm finds the most probable sequence of states for a given sequence of observations. The function inputs an observation sequence and returns the most probable sequence of states.

```
#Output of the Viterbi algorithm
test.viterbi(obs1)
```

```
['t', 't', 't', 't']
```

Baum-welch Algorithm

Using the principles of expectation maximization, the Baum-algorithm finds the emission, start and transition probabilities that represent a list of observation sequences. We use log-probability in order to prevent an overflow error. The function inputs as parameters, a set of observation sequences, the number of times each observation sequence occurs and the number of iterations. The function then returns the final emission, start and transition probabilities.

```
prob = test.log_prob(observation_tuple, quantities_observations)
print ("probability of sequence with original parameters : %f"%(prob))
```

```
probability of sequence with original parameters : -67.920122
```

```
#Sequence on which Baum welch algoritm aws applide on
print(observation_tuple)
print(quantities_observations)
```

```
[('A', 'B', 'B', 'A'), ('B', 'A', 'B')]
[10, 20]
```

```
#Apply Baum-welch Algorithm
num_iter=1000
emission,transition,start = test.train_hmm(observation_tuple,num_iter,quantities_
↪observations)
```

```
#Print output after applying the algorithm
print(emission)
```

```
[[ 0.36193015  0.63806985]
 [ 0.41111482  0.58888518]]
```

```
print(start)
```

```
[[ 0.49431308  0.50568692]]
```

```
print(transition)
```

```
[[ 0.58184591  0.41815409]
 [ 0.29789921  0.70210079]]
```

Notice that the probability of occurence of an observation sequence has increased for the new model parameters

```
prob = test.log_prob(observation_tuple, quantities_observations)
print ("probability of sequence after %d iterations : %f"%(num_iter,prob))
```

```
probability of sequence after 1000 iterations : -67.356668
```

Python Module Index

h

Index