EQ2341 - Assignment 3

Matay Mayrany - mayrany@kth.se

May 2022

1 Backward Algorithm

The backward algorithm complements the forward algorithm. The forward algorithm helps us reduce the computational complexity when evaluating the probability of a set of observations given a Hidden Markov Model. This is done by storing an alpha variable that helps us not repeat computations that are already done. This alpha variable tells us the probability of being at a current state i, at time t, given the observations we have seen so far. This does not consider the observations we have seen after t, which is where the backward algorithm. The backward algorithm helps correct our probability calculation by storing a variable, beta, which tells us the probability of being at state i at time t, given the observations that we will see in the future (after time t). It works recursively starting at the last observation, unlike the forward algorithm which works recursively starting with the first observation.

The instructions in the book were followed to implement the code needed for this algorithm. The code that was used to implement and test the backward algorithm can be seen in the sections below.

1.1 Formulae

Initialization step formulae from the text book:

Initialization: At $t = T$ we define, for reasons to be evident later,

for an infinite-duration HMM:

$$\beta_{i,T} = 1; \qquad \hat{\beta}_{i,T} = 1/c_T \tag{5.64}$$

for a finite-duration HMM λ:

$$\beta_{i,T} = a_{i,N+1}; \quad \hat{\beta}_{i,T} = \beta_{i,T}/(c_T c_{T+1}) \tag{5.65}$$

Backward step formula from the text book:

Backward step: The actual calculations use only the scaled backward variable, recursively updated as

$$\hat{\beta}_{i,t} = \frac{1}{c_t} \sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{x}_{t+1}) \hat{\beta}_{j,t+1}, \quad t = T-1, T-2, \ldots, 1 \tag{5.66}$$

1.2 Implementation

Implementation of initialization and backward steps:

```
def backward(self, scaledProbOfObservations, observations, c):
    T = scaledProbOfObservations.shape[1]
    J = self.A.shape[0]
    beta = np.zeros((J, T))
    one = np.ones(J)

    #Initialization Step
    if self.is_finite:
        beta[:, T - 1] = self.A[:, self.A.shape[1] - 1] / (c[T - 1] * c[T])
    else:
        beta[:, T - 1] = one / c[T - 1]


    #Backward Step
    for t in range(T - 2, -1, -1): #Starting with T-1 at index T - 2
        for i in range(J):
            probThatiCameBeforej = 0
            for j in range(J):
                probThatiCameBeforej += self.A[i, j] * beta[j, t + 1] * scaledProbOfObservations[j, t + 1]
            beta[i, t] += probThatiCameBeforej
        beta[:, t] = beta[:, t] / c[t]


    return beta
```

2 Tests and Verification of Implementation

In order to verify the implementation the examples given in the assignment descriptions were created in the last assignment, hence the code was reused and tweaked to test the backward algorithm. Code for the testing and screenshots of the results can be seen below.

2.1 Finite Duration Chain

Code for testing the backward algorithm for an HMM with a finite chain as described in the assignment:

```
from PattRecClasses import GaussD, HMM, MarkovChain
import numpy as np

#Test Backward algorithm FINITE CHAIN
finiteDurationMC = MarkovChain(np.array([1, 0]), np.array([[0.9, 0.1, 0], [0, 0.9, 0.1]]))

g1 = GaussD( means=[0], stdevs=[1] )   # Distribution for state = 1
g2 = GaussD( means=[3], stdevs=[2] )   # Distribution for state = 2
finiteDurationHMM  = HMM(finiteDurationMC, [g1, g2])
observations = np.array([-0.2, 2.6, 1.3])
scaledProbOfObservations = finiteDurationHMM.prob(observations, True)
alpha, c = finiteDurationMC.forward(scaledProbOfObservations)
c = [1, 0.1625, 0.8266, 0.0581]
print("C: ", c)
beta = finiteDurationMC.backward(scaledProbOfObservations, observations, c)
print("Values for finite duration HMM:")
print("betaHat:")
print(np.around(beta, 4))
```

Results:

```
C:  [1, 0.1625, 0.8266, 0.0581]
Values for finite duration HMM:
betaHat:
[[1.0003 1.0393 0.    ]
 [8.4182 9.3536 2.0822]]
```

Note That the results match what is described in the textbook, the reason the c values are hard coded is because using the direct values from the forward algorithm directly gave a slightly different values, due to the increased precision of the c values from the forward algorithm rather than the 4 decimal figures in the book. I could have rounded the output to 4 figures also.

2.2 Infinite Duration Chain

Code for testing the HMM with a infinite chain:

```
#Test Forward algorithm NON-FINITE CHAIN
nonfiniteDurationMC = MarkovChain( np.array([1, 0]), np.array([[0.9, 0.1], [0.1, 0.9]]))
nonfiniteDurationHMM  = HMM(nonfiniteDurationMC, [g1, g2])
scaledProbOfObservations = nonfiniteDurationHMM.prob(observations, True)
alpha, c = nonfiniteDurationMC.forward(scaledProbOfObservations)
print("C: ", c)
beta = nonfiniteDurationMC.backward(scaledProbOfObservations, observations, c)
print("Values for infinite duration HMM:")
print("betaHat:")
print(np.around(beta, 4))
```

Results:

```
C:  [1.         0.16252347 0.88811053]
Values for infinite duration HMM:
betaHat:
[[1.     6.7973 1.126 ]
 [5.2223 5.7501 1.126 ]]
```

3 Notes

Code used to answer all questions can be found in its respective section in the jupyter notebook and MarkovChain python classes.