EQ2341 Pattern Recognition and Machine Learning
Assignment 2

Boyue Jiang
boyue@kth.se

Ziyue Yang
ziyuey@kth.se

May 04, 2024

1 Introduction

In this project, our focus is on the implementation and validation of forward and backward algorithms in the class `MarkovChain` of the `PattRecClasses` framework designed for Hidden Markov Models (HMM). Our task is based on the mathematical foundations provided in the last chapters of the textbook, where these algorithms are described in detail.

The forward algorithm is important for calculating the probability of observing a particular sequence to a certain point given the initial model parameters. The process consists of three main steps: initialization, recursive computation, and termination.

In contrast, the backward algorithm calculates the probability of observing future segments of the sequence from a given state at a given time. This is crucial for the backward computation of the HMM, which, in combination with the results of the forward computation, helps in parameter estimation and sequence prediction.

This report will illustrate each step of the implementation, supported by code snippets and execution results to validate the effectiveness of our algorithms when modeling time series data using HMM.

2 Implementation Details

2.1 Forward Algorithm

The purpose of Forward algorithm is to compute $\hat{\alpha}_{j,t} = P[S_t = j \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_t, \lambda]$, denoted as `alpha_hat`, which is the probability of the state equals to j at time step `t` given all observations, and $c_t = P[\boldsymbol{x}_t \mid \boldsymbol{x}_1 \cdots \boldsymbol{x}_{t-1}, \lambda]$, denoted as `c`, the probability of an observation at `t` given past observations.

The Forward algorithm begins with an initialization phase where it computes the initial probabilities of each state, scaled by the first observation's likelihood. This process involves multiplying the initial state distribution `self.q` by the observation likelihood `pX[:,0]`, where `pX` is the matrix of emission probabilities corresponding to each observation at each time step. The author write a function to compute pX, which is stored in `'Assignment_2\PattRecClasses\func.py'`. The sum of these products gives the scaling factor `c[0]`, which is used to normalize the probabilities to prevent underflow during computation.

The main recursive computation then takes over from the second time step. For each subsequent time step

t, the algorithm calculates the probabilities of transitioning from all previous states to the current state, adjusted by the observation probability at time t. This involves matrix multiplication between the transpose of the transition matrix self.A and the previously computed alpha_hat[:,t-1], followed by element-wise multiplication with pX[:,t]. The result is normalized using the calculated scaling factor c[t] to ensure numerical stability.

```
def forward(self, pX):
# Initialization
    n, t_max = np.shape(pX) # n <=> # of states; t_max <=> sequence length
    c = np.zeros(t_max+1)
    alpha_hat = np.zeros([n,t_max])
    alpha_temp = self.q * pX[:,0] # 5.42
    c[0] = np.sum(alpha_temp) # 5.43
    alpha_hat[:,0] = alpha_temp / c[0] # 5.44

# Forward Step
    for t in range(1, t_max):
        alpha_temp = np.array([])
        for j in range(n):
            alpha_update = pX[j,t] * ( alpha_hat[:,t-1] @ self.A[:,j] ) # 5.50
            alpha_temp = np.append(alpha_temp, alpha_update) # 5.50
        c[t] = np.sum(alpha_temp, axis=0) # 5.51
        alpha_hat[:,t] = alpha_temp / c[t] # 5.52

# Termination
    if self.is_finite:
        c[-1] = alpha_hat[:,-1] @ self.A[:,-1] # 5.53
    else:
        c = c[0:-1] # length of c is different in infinite and finite case

    return alpha_hat, c
```

2.2 Backward Algorithm

The purpose of Backward algorithm is to compute $\hat{\beta}_{j,t}$, denoted as beta_hat, the scaled version of the probability of observing the sequence of observed states from time $t+1$ to T given that the system is in state j at time t.

The Backward algorithm initializes by setting the terminal probabilities. If the model is finite-duration (determined by self.is_finite), the final beta values beta_end are calculated by normalizing the transitions into the final state. For infinite-duration models, beta_end is simply set to a uniform distribution normalized by the final scaling factor c[-1], the last element in vector c.

The main loop of the backward algorithm iterates backwards from the penultimate observation, recalculating the probability of each state leading to the end of the sequence. This is computed by multiplying the transition probabilities self.A[i,:] with the corresponding probabilities of the observations and future states (beta_hat and pX). Each probability is then scaled by the reverse-order scaling factor c[t].

```
def backward(self, pX, c):
# Initialization
    n, t_max = np.shape(pX)
    beta_hat = np.zeros([n, t_max])
```

```
        if self.is_finite:
            beta_end = self.A[:,-1] / (c[-1]*c[-2]) # 5.65
        else:
            beta_end = np.ones(self.A.shape[0]) / c[-1] # 5.64
        beta_hat[:,-1] = beta_end

    # Backward Step
        c = np.flip(c) # reverse vector c for simplicity
        c = c[2:] if self.is_finite else c[1:]
        for t in range(c.shape[0]):
            beta_hat_t = np.zeros(n)
            for i in range(n):
                beta_i_t = np.sum(self.A[i,:n] * pX[:,-1-t] * beta_hat[:n,-1-t]) # 5.69
                beta_hat_i_t = beta_i_t / c[t] # 5.70
                beta_hat_t[i] = beta_hat_i_t
            beta_hat[:,-2-t] = beta_hat_t # recursive update

        return beta_hat
```

3 Verification and Results

3.1 Testing Setup

The verification of the Forward and Backward algorithms was conducted using two types of Hidden Markov Model configurations: finite and infinite. These models were tested using synthetic data to ensure control over the conditions and to clearly evaluate the effectiveness of the algorithms.

3.2 Finite Case

In the finite case, the model is designed with specific end-state probabilities, allowing for the evaluation of the termination step in the Forward algorithm and special handling in the Backward algorithm.

3.2.1 Code Execution and Results

```
        # Finite HMM Configuration
        mc = MarkovChain( np.array( [ 1, 0 ] ), np.array( [ [0.9, 0.1, 0], [0, 0.9, 0.1] ] ) )
        g1 = GaussD( means=[0], stdevs=[1] )
        g2 = GaussD( means=[3], stdevs=[2] )
        observations = np.array([-0.2, 2.6, 1.3])
        distribution = [g1, g2]
        h = HMM(mc, distribution)
        pX = compute_pX(observations, distribution, scale=True)
        alpha_hat, c = mc.forward(pX)
        log_prob = h.logprob(observations)

        np.set_printoptions(precision=4)
        print('alpha_hat = \n', alpha_hat, '\n\nc = \n', c, '\n\nLog Probability = \n', log_prob)
```

```
c = np.array([1, 0.1625, 0.8266, 0.0581])
beta_hat = mc.backward(pX, c)
print('\nbeta_hat = \n', beta_hat)
```

Alpha-hat Output:

```
[[0.3847 0.4189]
[0.6153 0.5811]]
```

C Output:

```
[1. 0.1625 0.8266 0.0581]
```

Beta-hat Output:

```
[[1.0003 1.0393 0.]
[8.4182 9.3536 2.0822]]
```

The finite case results show that the model correctly adjusts the state probabilities over time and scales them appropriately. The beta-hat values indicate the backward probabilities normalized by the scaling factors.

3.3 Infinite Case

In the infinite case, the model does not incorporate specific end-state transitions, which influences how probabilities are handled toward the sequence's end.

3.3.1 Code Execution and Results

```
mc = MarkovChain( np.array( [ 1, 0 ] ), np.array( [ [0.9, 0.1, 0], [0, 0.9, 0.1] ] ) )
g1 = GaussD( means=[0], stdevs=[1] )
g2 = GaussD( means=[3], stdevs=[2] )
observations = np.array([-0.2, 2.6, 1.3])
distribution = [g1, g2]
h = HMM(mc, distribution)
pX = compute_pX(observations, distribution, scale=True)
alpha_hat, c = mc.forward(pX)
log_prob = h.logprob(observations)

np.set_printoptions(precision=4)
print('alpha_hat = \n', alpha_hat, '\n\nc = \n', c, '\n\nLog Probability = \n', log_prob)

beta_hat = mc.backward(pX, c)
print('\nbeta_hat = \n', beta_hat)
```

Alpha-hat Output:

```
[[0.3847 0.4591]
 [0.6153 0.5409]]
```

C Output:

```
[1. 0.1625 0.8881]
```

Beta-hat Output:

```
[[1. 6.7973 1.126]
 [5.2223 7.7501 1.126]]
```

The results for the infinite case illustrate how the probabilities are handled continuously, with beta-hat values reflecting the increased uncertainty as more future observations are considered.