EQ2341 Pattern Recognition and Machine Learning
Project 2: Forward and Backward algorithm

Letao Feng
letao@kth.se

Rao Xie
raox@kth.se

1 Forward Algorithm

1.1 Formula

The algorithm calculates the conditional state probabilities given an observed feature sequence $(x_1, ..., x_t, ...)$ and a hidden Markov model (HMM) λ as follows:

$$\hat{\alpha}_{j,t} = P(S_t = j|\mathbf{x}_1, ..., \mathbf{x}_t, \lambda) \tag{1}$$

The Forward Algorithm consists of three steps, including initialization, forward steps, and termination(only needed for finite HMM).
For initialization, we have:

$$\alpha_{j,1}^{temp} = P[\boldsymbol{X}_1 = \boldsymbol{x}_1, S_1 = j \mid \lambda] = q_j b_j(\boldsymbol{x}_1), \quad j = 1 \dots N \tag{5.42}$$

$$c_1 = \sum_{k=1}^{N} \alpha_{k,1}^{temp} \tag{5.43}$$

$$\hat{\alpha}_{j,1} = \alpha_{j,1}^{temp} / c_1, \quad j = 1 \dots N \tag{5.44}$$

For forward steps:

$$\alpha_{j,t}^{temp} = b_j(\boldsymbol{x}_t) \left(\sum_{i=1}^{N} \hat{\alpha}_{i,t-1} a_{ij} \right), \quad j = 1 \dots N \tag{5.50}$$

$$c_t = \sum_{k=1}^{N} \alpha_{k,t}^{temp} \tag{5.51}$$

$$\hat{\alpha}_{j,t} = \alpha_{j,t}^{temp} / c_t, \quad j = 1 \dots N \tag{5.52}$$

For termination(only needed for finite HMM):

$$\begin{aligned} c_{T+1} &= P[S_{T+1} = N+1 \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_T, \lambda] \\ &= \sum_{k=1}^{N} P[S_T = k \cap S_{T+1} = N+1 \mid \boldsymbol{x}_1 \dots \boldsymbol{x}_T, \lambda] \\ &= \sum_{k=1}^{N} \hat{\alpha}_{k,T} a_{k,N+1} \end{aligned} \tag{5.53}$$

Additionally, to calculate Probability of a Feature Sequence($P(\mathbf{X} = \mathbf{x}|\lambda)$), we have:

$$\ln P[\boldsymbol{x}_1 \dots \boldsymbol{x}_T \mid \lambda] = \sum_{t=1}^{T} \ln c_t \tag{5.54}$$

1.2 Implementation

We completed the code exactly according to the formula described in 1.1.

```
def forward(self,pX):
    T = pX.shape[1]
    N = self.A.shape[0]
    c = np.zeros(T)
    temp = np.zeros((N, T))
    alfaHat = np.zeros((N, T))

    # Initialization
    for i in range(0, N):
        temp[i, 0] = self.q[i] * pX[i, 0]
        c[0] = c[0] + temp[i, 0]
    for i in range(0, N):
        alfaHat[i, 0] = temp[i, 0] / c[0]

    # Forward Steps
    for t in range(1, T):
        for j in range(0, N):
            temp[j, t] = pX[j, t] * np.sum(alfaHat[:, t - 1] * self.A[:, j])
            c[t] = c[t] + temp[j, t]
        for j in range(0, N):
            alfaHat[j, t] = temp[j, t] / c[t]

    # Termination
    if self.is_finite == True:
        tempc = c
        c = np.zeros(T + 1)
        c[:T] = tempc
        for i in range(0, N):
            c[T] = c[T] + alfaHat[i, T - 1] * self.A[i, N]

    return alfaHat, c
```

For the Probability of a Feature Sequence($P(\mathbf{X} = \mathbf{x}|\lambda)$), we have:

```
def logprob(self, x):
    pX_scaled = self.prob(x, False)
    alpha, c = self.stateGen.forward(pX_scaled)
    return np.sum(np.log(c))

def prob(self, x, shouldScale):
    x_number = len(x)
    b_number = len(self.outputDistr)
    pX = np.zeros((b_number, x_number))
    pX_scaled = np.zeros((b_number, x_number))

    for m in range(b_number):
        for n in range(x_number):
            pX[m, n] = self.outputDistr[m].prob(x[n])

    if shouldScale:
        max_vals = np.amax(pX, axis=0)
        pX /= max_vals


    return pX
```

1.3 Verification

According to Chapter A.3.1, the model parameters and observed feature sequences are as follows:

$$q = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0 & 0.9 & 0.1 \end{pmatrix}; \quad B = \begin{pmatrix} g_1(x) \\ g_2(x) \end{pmatrix}$$

where $g_1(x)$ and $g_2(x)$ follow a Gaussian distribution:

$$g_1(x) \sim \mathcal{N}(\mu_1 = 0, \sigma_1^2 = 1^2); \qquad g_2(x) \sim \mathcal{N}(\mu_2 = 3, \sigma_2^2 = 2^2)$$

Then we can get the result with the following code:

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

#Test Forward algorithm FINITE CHAIN
mc = 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
h  = HMM(mc, [g1, g2])
x = np.array([-0.2, 2.6, 1.3])

pX_scaled = h.prob(x,True)
alfaHat, c = mc.forward(pX_scaled)

logP=h.logprob(x)

print("alfaHat:")
print(np.around(alfaHat, 4))
print("c: ")
print(np.around(c, 4))
print("logP: ", logP)
```

```
alfaHat:
[[1.     0.3847 0.4189]
 [0.     0.6153 0.5811]]
c:
[1.     0.1625 0.8266 0.0581]
logP:  -9.187726979475208
```

As shown in the figure, the results are consistent with those written in the textbook, which shows that our code works correctly and successfully implements the forward algorithm.

2 Backward Algorithm

2.1 Formula

In the backward algorithm, the purpose is to find the backward variable:

$$\beta_{i,t} = P(X_{t+1} = x_{t+1}, ..., X_t = x_t | S_t = i, \lambda) \tag{2}$$

for an infinite-duration HMM, or

$$\beta_{i,t} = P(X_{t+1} = x_{t+1}, ..., X_t = x_t, S_{t+1} = N+1 | S_t = i, \lambda) \tag{3}$$

for a finite-duration HMM.
Similar to the forward algorithm, the procedure has two steps, including the initialization and the backward step.
For initialization:

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:

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

For Backward step:

$$\begin{aligned} \hat{\beta}_{i,t} &= \beta_{i,t}/(c_t \cdots c_T c_{T+1}) \\ &= \frac{1}{c_t} \sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{x}_{t+1}) \hat{\beta}_{j,t+1} \end{aligned} \tag{5.70}$$

2.2 Implementation

We completed the code exactly according to the formula described in 2.1.

```
def backward(self, c, pX):
    T = pX.shape[1]
    N = self.A.shape[0]
    betaHat = np.zeros((N, T))

    # Initialization
    if self.is_finite == False:
        for i in range(0, N):
            betaHat[i, T-1] = 1/c[T-1]
    else:
        for i in range(0, N):
            betaHat[i, T-1] = self.A[i, N]/(c[T]*c[T-1])

    # Backward Step
    if self.is_finite == True:
        self.A = self.A[:, :N]

    for t in range(T-1, 0, -1):
        for i in range(0, N):
            betaHat[i, t-1] = np.sum(self.A[i, :] * pX[:, t] * betaHat[:, t]) / c[t-1]

    return betaHat
```

2.3 Verification

According to Chapter A.3.2, the model parameters and observed feature sequences are as follows:

$$q = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0 & 0.9 & 0.1 \end{pmatrix}; \quad B = \begin{pmatrix} g_1(x) \\ g_2(x) \end{pmatrix}$$

where $g_1(x)$ and $g_2(x)$ follow a Gaussian distribution:

$$g_1(x) \sim \mathcal{N}(\mu_1 = 0, \sigma_1^2 = 1^2); \qquad g_2(x) \sim \mathcal{N}(\mu_2 = 3, \sigma_2^2 = 2^2)$$

Then we can get the result with the following code:

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

#Test Forward algorithm FINITE CHAIN
mc = 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
h  = HMM(mc, [g1, g2])
x = np.array([-0.2, 2.6, 1.3])

pX_scaled = h.prob(x,True)

c_answer = [1, 0.1625, 0.8266, 0.0581]
betaHat=mc.backward(c_answer, pX_scaled)

print("betaHat: ")
print(np.around(betaHat, 4))
```

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

As shown in the figure, the results are consistent with those written in the textbook, which shows that our code works correctly and successfully implements the backward algorithm.