<u>Hidden Markov Model</u>

By

Charlie Silkin

Contents

PROJECT INSTRUCTIONS

1. The dataset `hmm_pb1.csv` represents a sequence of dice rolls x from the Dishonest casino model. The states of Y are 1='Fair' and 2='Loaded'.
   a) Implement the Viterbi algorithm and find the most likely sequence y that generated the observed x. Use the log probabilities, as shown in the HMM slides. Report the obtained sequence y of 1's and 2's for verification.
   b) Implement the forward and backward algorithms and run them on the observed x. You should memorize a common factor u_t for the α_t^k to avoid floating point underflow, since α_t^k quickly become very small. The same holds for β_k^t. Report $\frac{\alpha_{133}^1}{\alpha_{133}^2}$ and $\frac{\beta_{133}^1}{\beta_{133}^2}$, where the counting starts from $t = 0$.

2. The dataset `hmm_pb2.csv` represents a sequence of 10000 dice rolls x from the Dishonest casino model but with other values for the a and b parameters. Having so many observations, you are going to learn the model parameters. Implement and run the Baum-Welch algorithm using the forward and backward algorithms that you already implemented. You can initialize the π, a, b with your guess, or with some random probabilities (make sure that π sums to 1 and that a_i^j , b_i^k sum to 1 for each i). The algorithm converges quite slowly, so you might need to run it for up 1000 iterations or more for the parameters to converge. Report the values of π, a, b that you have obtained.

STARTUP CODE (LOAD PACKAGES AND DATA)

```
# Import packages:

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import torch


# Load hmm_pb1.csv

x = np.loadtxt("C:/Users/Student/Desktop/FSU -- Data Science Master's/2023
Spring/STA 5635 -- Applied Machine Learning/hmm_pb1.csv", delimiter = ",")

print(x)

# Load hmm_pb2.csv

x2 = np.loadtxt("C:/Users/Student/Desktop/FSU -- Data Science
Master's/2023 Spring/STA 5635 -- Applied Machine Learning/hmm_pb2.csv",
delimiter = ",")

print(x2)
```

PART 1.A (VITERBI ALGORITHM)

RESULTS

$y =$ [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]

CODE

```
# Viterbi algorithm:
def Viterbi(x,a,b,pi):
    ## Initialize i and k
    i = x.shape[0]
    k = a.shape[0]

    ## Create C matrix
    C = np.zeros((i, k))

    ## Initialize first row of C
    b_0 = b[:, int(x[0])-1]

    C[0,:] = np.log(pi * b_0)

    ## Create matrix of most probable hidden states
    prev = np.zeros((i-1, k))
```

```
    ## Iterate over time range
    for t in range(1, i):
        for j in range(k):
            ### Obtain probability values
            b_xt = b[j, int(x[t])-1]
            probability = C[t-1] + np.log(a[:,j]) + np.log(b_xt)

            # Update matrix of most probable hidden states
            prev[t-1, j] = np.argmax(probability)

            # Update row t of C matrix with probability of the most
probable state
            C[t,j] = np.max(probability)

    ## Array of sequence
    S = np.zeros(i)

    ## Find the most probable last hidden state
    last_state = np.argmax(C[i-1, :])

    S[0] = last_state

    ## Trace back through most probable hidden states to get full y*
    backtrack_index = 1
    for m in range(i-2, -1, -1):
        S[backtrack_index] = prev[m, int(last_state)]
        last_state = prev[m, int(last_state)]
        backtrack_index += 1

    ### Flip sequence array since we were backtracking
    S = np.flip(S, axis=0)
```

```
    ### Replace state values with "1" for fair, "2" for loaded
    y = []
    for s in S:
        if s == 0:
            y.append(1)
        else:
            y.append(2)

    ### Return final sequence
    return y

# PART 1.A RESULT:
for yi in Viterbi(x,a,b,pi):
    print(yi, end = ",")
```

PART 1.B (FORWARD AND BACKWARD ALGORITHMS)

RESULTS

$$\frac{\alpha_{133}^{1}}{\alpha_{133}^{2}} = \frac{0.84}{0.16} = 5.26$$

$$\frac{\beta_{133}^{1}}{\beta_{133}^{2}} = \frac{0.223}{0.777} = 0.287$$

CODE

```
# Forward algorithm:
def forward_algorithm(x,a,b,pi):

    ## Initialize i and k
    i = x.shape[0]
    k = a.shape[0]

    ## Create matrix of alphas
    alpha_mat = np.zeros((i, k))

    ## Initialize first row of alpha matrix
    b_0 = b[:, int(x[0])-1]
    alpha_mat[0,:] = (pi * b_0)

    ## Normalize alphas so that probabilities of each row sum to 1
    alpha_mat[0,:] = alpha_mat[0,:] / np.sum(alpha_mat[0,:], axis =
0)

    ## Obtain alpha values at time t
    for t in range(1, i):
        for j in range(k):
            b_xt = b[j, int(x[t])-1]
```

```
            alpha_mat[t,j] = b_xt * np.dot(alpha_mat[t-1], a[:,j])
        ### Normalize alphas so that probabilities of each row sum
to 1
        alpha_mat[t,:] = alpha_mat[t,:] / np.sum(alpha_mat[t,:],
axis = 0)

    ## Return alpha matrix
    return alpha_mat

# Backwards algorithm:
def backward_algorithm(x,a,b):

    ## Initialize i and k
    i = x.shape[0]
    k = a.shape[0]

    ## Create matrix of betas
    beta_mat = np.zeros((i, k))

    ## Initialize last row of betas as ones
    beta_mat[i-1] = np.ones((k))

    ## Normalize betas so that probabilities of each row sum to 1
    beta_mat[i-1] = beta_mat[i-1] / np.sum(beta_mat[i-1], axis = 0)

    ## Obtain betas at time t+1
    for t in range(i-2, -1, -1):
        for j in range(k):
            b_xt_plus1 = b[j, int(x[t+1])-1]
            beta_mat[t,j] = np.dot((beta_mat[t+1] * b_xt_plus1),
a[j,:])
```

```
        ### Normalize betas so that probabilities of each row sum to
1

        beta_mat[t,:] = beta_mat[t,:] / np.sum(beta_mat[t,:], axis =
0)


    ## Return beta matrix

    return beta_mat


# QUESTION 1b RESULTS:

alpha_133 = forward_algorithm(x,a,b,pi)[133]


alpha_rat = alpha_133[0] / alpha_133[1]


print(alpha_133[0], alpha_133[1], alpha_rat)


beta_133 = backward_algorithm(x,a,b)[133]


beta_rat = beta_133[0] / beta_133[1]


print(beta_133[0], beta_133[1], beta_rat)
```

PART 2 (BAUM-WELCH ALGORITHM)

RESULTS

$$\pi = [0.001 \quad .999]$$

$$a = \begin{bmatrix} 0.625 & 0.375 \\ 0.012 & 0.988 \end{bmatrix}$$

$$b = \begin{bmatrix} 0.079 & 0.11 & 0.062 & 0.039 & 0.625 & 0.084 \\ 0.201 & 0.205 & 0.193 & 0.201 & 0.107 & 0.094 \end{bmatrix}$$

CODE

```
# Initialize parameters with custom values

pi2 = [0.87,0.13]

a2 = np.array(((0.87,0.13), (0.13,0.87)))

b2 = np.array(((1/6,1/6,1/6,1/6,1/6,1/6),
(0.3,0.27,0.05,0.16,0.11,0.11)))


# Baum-Welch algorithm:

def Baum_Welch(x,a,b,pi,n_iter=1000):

    ## Initialize i and k

    i = x.shape[0]

    k = a.shape[0]


    ## Iterations

    for n in range(n_iter):


        ### Initialize alphas and betas

        alpha = forward_algorithm(x,a,b,pi)

        beta = backward_algorithm(x,a,b)
```

```
        ### Create xi matrix
        xi = np.zeros((k,k,i-1))

        ### Compute appropriate values of xi
        for t in range(i-1):
            b_xt_plus1 = b[:, int(x[t+1])-1]
            denom_pt1 = np.dot(alpha[t,:], a) * b_xt_plus1
            denom = np.dot(denom_pt1,beta[t+1,:])

            for j in range(k):
                alpha_A = alpha[t,j] * a[j,:]

                beta_B = b_xt_plus1 * beta[t+1,:]

                num = alpha_A * beta_B

                xi[j,:,t] = num/denom

        ## Create gamma matrix
        gamma = np.sum(xi, axis = 1)

        ## Update pi as gamma values at time t=1
        pi = gamma[:,1]/np.sum(gamma[:,1])

        ## Update a
        a_num = np.sum(xi, 2) ### Normalization to ensure horizontal
sums of 1
        a_denom = np.sum(gamma, axis=1).reshape((-1, 1))

        a = a_num / a_denom
```

```
        ## Add element to end of gamma to match dimensions

        gamma = np.hstack((gamma, np.sum(xi[:,:,i-2],
axis=0).reshape((-1, 1))))


        ## Update b

        b_denom = np.sum(gamma, axis=1).reshape((-1, 1)) ###
Normalization to ensure horizontal sums of 1


        for num in range(b.shape[1]):

            b[:, num] = np.sum(gamma[:, x == num+1], axis=1)


        b = b/b_denom


    ## Return final parameters

    return (pi, a, b)


# QUESTION 2 RESULTS:

print(Baum_Welch(x2,a2,b2,pi2))
```

REFERENCES

1. https://github.com/adeveloperdiary/HiddenMarkovModel/tree/master/part4
2. https://www.youtube.com/watch?v=6JVqutwtzmo
3. https://numpy.org/doc/stable/reference/generated/numpy.dot.html
4. https://stackoverflow.com/questions/8437964/python-printing-horizontally-rather-than-current-default-printing