Jarod Klion

STA5635

March 23rd, 2022

1.

a. Most likely sequence *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

b. $\alpha^1_{128} / \alpha^2_{128} = 1.634322846$

c. $\beta^1_{128} / \beta^2_{128} = 0.800134907$

2. $\pi = \begin{bmatrix} 0.66354372 & 0.33645628 \end{bmatrix}$

$a = \begin{bmatrix} 0.53691227 & 0.46308773 \\ 0.47678048 & 0.52321592 \end{bmatrix}$

$b = \begin{bmatrix} 0.1826396 & 0.2165343 & 0.1756936 & 0.2094443 & 0.1155081 & 0.1001801 \\ 0.2110478 & 0.1872504 & 0.2015701 & 0.1811477 & 0.1328683 & 0.0861157 \end{bmatrix}$

```
#!/usr/bin/env python
# coding: utf-8

# In[780]:



import numpy as np
import pandas as pd
import matplotlib.pyplot as plt



# # Viterbi

# In[782]:



# From "https://stackoverflow.com/questions/9729968/python-implementation-of-viterbi-algorithm"
def viterbi(y, A, B, Pi=None):
    """
    Return the MAP estimate of state trajectory of Hidden Markov Model.

    Parameters
    ----------
    y : array (T,)
        Observation state sequence. int dtype.
    A : array (K, K)
        State transition matrix. See HiddenMarkovModel.state_transition  for
        details.
    B : array (K, M)
        Emission matrix. See HiddenMarkovModel.emission for details.
    Pi: optional, (K,)
        Initial state probabilities: Pi[i] is the probability x[0] == i. If
        None, uniform initial distribution is assumed (Pi[:] == 1/K).
```

```
    Returns
    -------
    x : array (T,)
        Maximum a posteriori probability estimate of hidden state trajectory,
        conditioned on observation sequence y under the model parameters A, B,
        Pi.
    T1: array (K, T)
        the probability of the most likely path so far
    T2: array (K, T)
        the x_j-1 of the most likely path so far
    """

    # Cardinality of the state space
    K = A.shape[0]
    # Initialize the priors with default (uniform dist) if not given by caller
    Pi = Pi if Pi is not None else np.full(K, 1 / K)
    T = len(y)
    T1 = np.empty((K, T), 'd')
    T2 = np.empty((K, T), 'B')

    # Initialize the tracking tables from first observation
    T1[:, 0] = np.log(Pi * B[:, y[0]])
    T2[:, 0] = 0

    # Iterate through the observations updating the tracking tables
    for i in range(1, T):
        #need to subtract 1 from y as it ranges from 1-6 but indexed 0 in array
        T1[:, i] = np.max(T1[:, i - 1] + np.log(A), 1) + np.log(B[np.newaxis, :, y[i] -
1])
        T2[:, i] = np.argmax(T1[:, i - 1] + np.log(A), 1)

    # Build the output, optimal model trajectory
    x = np.empty(T, 'B')
```

```
    x[-1] = np.argmax(T1[:, T - 1])
    #flip since we are backtracking
    for i in reversed(range(1, T)):
        x[i - 1] = T2[x[i], i]

    return x, T1, T2


# In[783]:


obs_a = np.loadtxt("../datasets/markov/hmm_pb1.csv", dtype = int, delimiter = ",")
obs_a_pd = pd.read_csv("../datasets/markov/hmm_pb1.csv", header=None)
pi_a = np.array([0.5, 0.5])
trans_a = np.array([
                    [0.95, 0.05],
                    [0.05, 0.95]
                   ])
emit_a = np.array([
                    [1/6, 1/6, 1/6, 1/6, 1/6, 1/6],
                    [1/10, 1/10, 1/10, 1/10, 1/10, 1/2]
                  ])


# In[1186]:


y, T1, T2 = viterbi(obs_a, trans_a, emit_a, pi_a)


# In[1188]:


#y = 1 is Fair, y=2 is loaded
```

```
print(y+1)

print(T1)

print(T2)




# # Forward Algorithm


# In[1216]:




#Modified from "http://www.adeveloperdiary.com/data-science/machine-learning/forward-and-backward-algorithm-in-hidden-markov-model/"

def forward(observations, transition_matrix, emission_matrix, pi):

    #python indexed 0 so subtract 1 from all dice rolls

    observations = observations - 1

    #get number of observations

    observations_rows = observations.shape[0]

    #get number of states (K)

    K = transition_matrix.shape[0]


    #Initialize alpha probabilities

    alpha = np.zeros([observations_rows, K])

    alpha[0, :] = pi * emission_matrix[:, observations[0]]

    alpha_probs = np.zeros([observations_rows, K])

    alpha_probs[0] = alpha[0] / np.sum(alpha[0])


    #common factor to avoid underflow

    #u = np.zeros([observations_rows - 1, K])


    #iterate forward through observations to compute alpha probabilities

    for t, obs in enumerate(observations[1:], 1):

            alpha[t] = np.array([

                            #start with probability we go from previous state to fair

                            emission_matrix[0][obs] * np.sum([
```

```
                    #chance from fair to fair
                    alpha[t - 1][0] * transition_matrix[0, 0],
                    #chance from loaded to fair
                    alpha[t - 1][1] * transition_matrix[1, 0]
                    ]),
                #go from previous state to loaded
                emission_matrix[1][obs] * np.sum([
                    #chance from fair to loaded
                    alpha[t - 1][0] * transition_matrix[0, 1],
                    #chance from loaded to loaded
                    alpha[t - 1][0] * transition_matrix[1, 1]
                    ]),
                ])
alpha_probs[t] = np.array([
                #start with probability we go from previous state to fair
                emission_matrix[0][obs] * np.sum([
                    #chance from fair to fair
                    alpha_probs[t - 1][0] * transition_matrix[0, 0],
                    #chance from loaded to fair
                    alpha_probs[t - 1][1] * transition_matrix[1, 0]
                    ]),
                #go from previous state to loaded
                emission_matrix[1][obs] * np.sum([
                    #chance from fair to loaded
                    alpha_probs[t - 1][0] * transition_matrix[0, 1],
                    #chance from loaded to loaded
                    alpha_probs[t - 1][0] * transition_matrix[1, 1]
                    ]),
                ])
#normalize for probabilities
alpha_probs[t] /= np.sum(alpha_probs[t])
#still need u_t for underflow ?
#u[t - 1, :] = alpha[t] / alpha[t - 1]
```

```
    print(alpha[128][0] / alpha[128][1])

    return alpha_probs



# In[1214]:



alpha = forward(obs_a, trans_a, emit_a, pi_a)



# In[1215]:



print(alpha)



# # Backwards Algorithm

# In[1217]:




#Modified from from "http://www.adeveloperdiary.com/data-science/machine-
learning/forward-and-backward-algorithm-in-hidden-markov-model/"

def backward(observations, transition_matrix, emission_matrix):

    #python indexed 0 so subtract 1 from all dice rolls

    observations = observations - 1

    #get number of observations

    observations_rows = observations.shape[0]

    #get number of states

    K = transition_matrix.shape[0]


    #initial probabilities for beta start at T and work backwards

    beta, beta_probs = np.zeros([observations_rows, K]), np.zeros([observations_rows,
K])
```

```
beta[observations_rows - 1] = np.array([1, 1])
beta_probs[observations_rows - 1] = np.array([1, 1])

for t, obs in reversed(list(enumerate(observations))[1:]):
    beta[t - 1] = np.array([
        np.sum([
            #from fair to fair
            transition_matrix[0, 0] * emission_matrix[0][obs] * beta[t][0],
            #from fair to loaded
            transition_matrix[0, 1] * emission_matrix[1][obs] * beta[t][1],
        ]),
        np.sum([
            #from loaded to fair
            transition_matrix[1, 0] * emission_matrix[0][obs] * beta[t][0],
            #from loaded to loaded
            transition_matrix[1, 1] * emission_matrix[1][obs] * beta[t][1]
        ])
    ])
    beta_probs[t - 1] = np.array([
        np.sum([
            #from fair to fair
            transition_matrix[0, 0] * emission_matrix[0][obs] * beta_probs[t][0],
            #from fair to loaded
            transition_matrix[0, 1] * emission_matrix[1][obs] * beta_probs[t][1],
        ]),
        np.sum([
            #from loaded to fair
            transition_matrix[1, 0] * emission_matrix[0][obs] * beta_probs[t][0],
            #from loaded to loaded
            transition_matrix[1, 1] * emission_matrix[1][obs] * beta_probs[t][1]
        ])
    ])

    #normalize beta values
```

```
        beta_probs[t - 1] /= np.sum(beta_probs[t - 1])
    print(beta[128][0] / beta[128][1])

    return beta_probs


# In[1218]:


beta = backward(obs_a, trans_a, emit_a)


# In[1219]:


print(beta)


# # 2. Baum Welch Algorithm

# In[958]:


obs_b = pd.read_csv("../datasets/markov/hmm_pb2.csv", header=None).values
obs_b = obs_b[0]
trans_EM = np.random.rand(2, 2)
trans_EM /= trans_EM.sum(axis=1)[:, None]
emit_EM = np.random.rand(2, 6)
emit_EM /= emit_EM.sum(axis=1)[:, None]
pi_EM = np.random.rand(2)
pi_EM /= pi_EM.sum()

#Check initial guesses
print(trans_EM)
```

```
print(emit_EM)
print(pi_EM)
#Check their probabilities add to 1
print(trans_EM.sum(axis=1))
print(emit_EM.sum(axis=1))
print(np.sum(pi_EM))
```

```
# In[1221]:
```

```
def baum_welch(observations, transition_matrix, emission_matrix, pi, epochs = 50):

    observations = observations - 1
    observations_rows = observations.shape[0]
    #number of states
    K = transition_matrix.shape[0]

    for epoch in range(epochs):
        #initialize placeholders for observations
        new_transition_matrix = np.zeros((2, 2))
        new_pi = np.zeros(2)
        new_emission_matrix = np.zeros((2, 6))
        fitness = 0

        alphas = forward(observations, transition_matrix, emission_matrix, pi)
        betas = backward(observations, transition_matrix, emission_matrix)

        #collect probabilities of each observation
        probability_of_obs = pi[0] * betas[0][0] + pi[1] * betas[0][1]
        fitness += probability_of_obs

        #xi calculations - probability of being in state 'i' at time 't'
        #and state 'j' at time 't+1' based on the model
```

```
        xi = np.zeros([observations_rows, K, K])
        for t in range(observations_rows - 1):
            denominator = np.dot(np.dot(alphas[t, :].T, transition_matrix) *
emission_matrix[:, observations[t + 1]].T, betas[t + 1, :])
            xi[t, 0, 0] = (alphas[t][0] * transition_matrix[0, 0]
                          * emission_matrix[0][observations[t + 1]] * betas[t + 1][0]) /
denominator
            xi[t, 0, 1] = (alphas[t][0] * transition_matrix[0, 1]
                          * emission_matrix[1][observations[t + 1]] * betas[t + 1][1]) /
denominator
            xi[t, 1, 0] = (alphas[t][1] * transition_matrix[1, 0]
                          * emission_matrix[0][observations[t + 1]] * betas[t + 1][0]) /
denominator
            xi[t, 1, 1] = (alphas[t][1] * transition_matrix[1, 1]
                          * emission_matrix[1][observations[t + 1]] * betas[t + 1][1]) /
denominator

        #gamma stores probability that we are in state 'i' at time t
        gamma = np.zeros([observations_rows, K])
        for t in range(observations_rows - 1):
            gamma[t, 0] = np.sum(xi[t, 0, :])
            gamma[t, 1] = np.sum(xi[t, 1, :])

        #recompute initial probabilities for the observation
        new_pi = gamma[0] / np.sum(gamma[0])

        # calculate new transition matrix for later update
        new_transition_matrix[0, 0] += np.sum(xi[:, 0, 0]) / np.sum(gamma[:, 0])
        new_transition_matrix[0, 1] += np.sum(xi[:, 0, 1]) / np.sum(gamma[:, 0])
        new_transition_matrix[1, 0] += np.sum(xi[:, 1, 0]) / np.sum(gamma[:, 1])
        new_transition_matrix[1, 1] += np.sum(xi[:, 1, 1]) / np.sum(gamma[:, 1])

        # calculate new emission matrix for later update
        emit_denom = np.sum(gamma, axis=0)
        for l in range(emission_matrix.shape[1]):
```

```
            #new_emission_matrix[:, l] = np.sum(gamma[observations == l, :]) /
emit_denom

            new_emission_matrix[0, l] = np.sum(np.take(gamma[:, 0],
np.where(observations == l))) / emit_denom[0]

            new_emission_matrix[1, l] = np.sum(np.take(gamma[:, 1],
np.where(observations == l))) / emit_denom[1]


        #update all values for probabilities for next epoch

        pi = new_pi

        transition_matrix = new_transition_matrix

        emission_matrix = new_emission_matrix



    # Some logging is always good :)

        print('= EPOCH #{} ='.format(epoch))

        print('Transition Matrix:', transition_matrix)

        print('Initial Dice Probability:', pi)

        print('Emission Matrix:', emission_matrix)

        print('Fitness:', fitness)

        print()



# In[1183]:



baum_welch(obs_b, trans_EM, emit_EM, pi_EM, epochs = 1000)
```