4) Below is the code for th forward backward algorithm and Viterbi algorithm:

import numpy as np

class HMM():

'''

Given a sequence, we use the forward backward algorithm to compute the most

likely state at each time and we compute the most likely sequence of states

using the viterbi algorithm

'''

def \_\_init\_\_(self, emit\_matrix, transfer\_matrix, sequence,

init\_distribution = np.array([0.5, 0.5])):

self.emit\_matrix = emit\_matrix

self.transfer\_matrix = transfer\_matrix

self.sequence = sequence

self.init\_distribution = init\_distribution

self.num\_states = len(init\_distribution)

self.observation = self.create\_obs(sequence)

def create\_obs(self, sequence):

obs = []

for event in sequence:

diag = []

for state in range(self.num\_states):

diag.append(self.emit\_matrix[state][event - 1])

obs.append(np.diag(diag))

obs = np.array(obs)

return obs

def forward(self, t):

# Calculate the forward probabilities

if t == 0:

return self.init\_distribution

else:

first = np.matmul(self.observation[t - 1], self.transfer\_matrix.T)

not\_normalalized = np.matmul(first, self.forward(t - 1))

norm = np.sum(not\_normalalized)

normalized = not\_normalalized / norm

return normalized

def backward(self, t):

# Calculate the backwrards probabilities

if t == len(self.sequence) + 1:

return np.ones([self.num\_states])

else:

first = np.matmul(self.transfer\_matrix, self.observation[t - 1])

not\_normalalized = np.matmul(first, self.backward(t + 1))

norm = np.sum(not\_normalalized)

normalized = not\_normalalized / norm

return normalized

def smoothing(self, t):

#Smoothing to find most likely state at any time

total\_prob = sum([self.forward(i) \* self.backward(i) for i in range(len(self.sequence) + 1)])

prob = self.forward(t) \* self.backward(t) / total\_prob

print(prob / sum(prob))

def path\_likelihood(self, path):

# Compute the log likelihood of a given path

likelihood = 0

likelihood += -np.log(self.init\_distribution[path[0]])

for i in range(1, len(path)):

transition\_log = -np.log(self.transfer\_matrix[path[i - 1], path[i]])

#print(i)

emit\_log = - np.log(self.emit\_matrix[path[i - 1], self.sequence[i - 1] - 1])

likelihood += transition\_log + emit\_log

return likelihood

def shortest\_path(self, time, state):

# Find the shortest path of length time ending in state

if time == 0:

return [state]

else:

options = []

min\_path = 0

for i in range(self.num\_states):

path = self.shortest\_path(time - 1, i) + [state]

options.append(self.path\_likelihood(path))

best\_choice = options.index(min(options))

best\_path = self.shortest\_path(time - 1, best\_choice) + [state]

print(best\_path)

return best\_path

def viterbi(self):

if self.path\_likelihood(self.shortest\_path(len(self.sequence), 1)) < self.path\_likelihood(self.shortest\_path(len(self.sequence), 0)):

return self.shortest\_path(len(self.sequence), 1)

else:

return self.shortest\_path(len(self.sequence), 0)