Jarod Klion

STA5635

March 23rd, 2022

* 1. 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

1. π =

a =

b =

#!/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)