##########

# HMM.py returns the firing rate selected as an alternative hidden state.

# needs libraries: (matplotlib, numpy, pandas).

# Instruction

# put HMM.py in a folder on a path.

# import HMM

# then you may obtain HMM.().

# you need only HMM function.

# the function HMM take a spike train as an argument.

# spike train could be given by list or numpy.array.

# parameters are determined by the HMM and a figure is drawn.

# references:

# Mochizuki and Shinomoto, Analog and digital codes in the brain

# https://arxiv.org/abs/1311.4035

# Contact:

# Shigeru Shinomoto: shinomoto@scphys.kyoto-u.ac.jp

##########

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

import math

def HMM(spike\_times) :

spike\_times = np.array(list(spike\_times))

max\_value = max(spike\_times)

min\_value = min(spike\_times)

onset = min\_value - 0.001 \* (max\_value - min\_value)

offset = max\_value + 0.001 \* (max\_value - min\_value)

bin\_width = (offset - onset) / len(spike\_times) \* 5

rate\_hmm = get\_hmm\_ratefunc(spike\_times, bin\_width, max\_value, min\_value)

drawHMM(spike\_times, rate\_hmm)

return rate\_hmm

####

# draws the rate of event occurrence.

# arguments:

# spike\_times: spike train

# rate\_hmm: estimated rate

####

def drawHMM(spike\_times, rate\_hmm) :

xaxis = [rate\_hmm[0, 0]]

yaxis = [rate\_hmm[0, 1]]

tempx\_old = tempx = rate\_hmm[0, 0]

tempy\_old = tempy = rate\_hmm[0, 1]

for i in range(0, len(rate\_hmm) - 1) :

tempx, tempy = rate\_hmm[i]

if (tempy != tempy\_old) :

mid = (tempx + tempx\_old) / 2

xaxis.append(mid)

xaxis.append(mid)

yaxis.append(tempy\_old)

yaxis.append(tempy)

tempx\_old = tempx

tempy\_old = tempy

xaxis.append(rate\_hmm[-1, 0])

yaxis.append(rate\_hmm[-1, 1])

plt.stackplot(xaxis, yaxis)

plt.xlim(xmin = min(xaxis), xmax = max(xaxis))

plt.ylim(ymin = 0)

plt.show()

####

# get\_hmm\_ratefunc

# estimate the rate of event occurrence with the EM algorithm.

# parameters to be determined: (matrix A, vector pi, vector lambda, matrix Gamma, vector Xi).

# arguments:

# spike\_times: spike train

# bin\_width: the bin is set to be 5 times of mean inter spike interval

# max\_value: time of the final spike

# min\_value: time of the initial spike

# returns the rate of event occurrence.

####

def get\_hmm\_ratefunc(spike\_times, bin\_width, max\_value, min\_value) :

EMloop\_num = 5000

mat\_A = np.array([[0.999, 0.001], [0.001, 0.999]])

vec\_pi = np.array([0.5, 0.5])

mean\_rate = len(spike\_times) / (max\_value - min\_value)

vec\_lambda = np.empty(2)

vec\_lambda[0] = (mean\_rate \* 0.75) \* bin\_width

vec\_lambda[1] = (mean\_rate \* 1.25) \* bin\_width

vec\_spkt = np.array([spike - min\_value for spike in spike\_times])

vec\_Xi = get\_vec\_Xi(vec\_spkt, bin\_width)

mat\_Gamma, mat\_Xi = HMM\_E\_step(vec\_Xi, mat\_A, vec\_lambda, vec\_pi)

mat\_A\_old = mat\_A.copy()

vec\_pi\_old = vec\_pi.copy()

vec\_lambda\_old = vec\_lambda.copy()

loop = 0

flag = 0

while(loop <= EMloop\_num and flag == 0) :

vec\_pi\_new, vec\_lambda\_new, mat\_A\_new = HMM\_M\_step(vec\_Xi, mat\_A, vec\_lambda, vec\_pi, mat\_Gamma, mat\_Xi)

vec\_pi = vec\_pi\_new.copy()

vec\_lambda = vec\_lambda\_new.copy()

mat\_A = mat\_A\_new.copy()

sum\_check = 0.0

num\_state = len(vec\_pi)

sum\_check += sum(abs(vec\_pi\_old - vec\_pi))

sum\_check += sum(abs(vec\_lambda\_old - vec\_lambda))

sum\_check += sum(sum(abs(mat\_A\_old - mat\_A)))

if (sum\_check / (1.0 \* num\_state \* (num\_state + 2)) < 1.0e-7) :

flag = 1

mat\_A\_old = mat\_A.copy()

vec\_pi\_old = vec\_pi.copy()

vec\_lambda\_old = vec\_lambda.copy()

E\_res = HMM\_E\_step(vec\_Xi, mat\_A, vec\_lambda, vec\_pi)

mat\_Gamma = E\_res[0]

mat\_Xi = E\_res[1]

loop += 1

vec\_hidden = HMM\_Viterbi(vec\_Xi, mat\_A, vec\_lambda, vec\_pi)

rate\_func = np.empty([len(vec\_Xi), 2])

c\_time = 0.0

for n in range(0, len(vec\_Xi)) :

state\_id = vec\_hidden[n]

rate\_func[n][0] = round(c\_time \* 100) / 100.0

rate\_func[n][1] = round(vec\_lambda[int(state\_id)] \* 100) / (bin\_width \* 100.0)

c\_time += bin\_width

return rate\_func

def get\_vec\_Xi(vec\_spkt, bin\_width) :

spkt\_dura = vec\_spkt[len(vec\_spkt) - 1]

bin\_num = int(math.ceil(spkt\_dura / bin\_width))

vec\_Xi = np.zeros(bin\_num)

for x in vec\_spkt:

bin\_id = int(math.floor(x / bin\_width))

if (bin\_id < bin\_num) :

vec\_Xi[bin\_id] += 1

return vec\_Xi

####

# HMM\_E\_step

# carries out the E step.

# arguments:

# vec\_Xi: numpy array class

# mat\_A: numpy array class

# vec\_lambda: numpy array class

# vec\_pi: numpy array class

# returns

# mat\_Gamma: numpy array class

# mat\_Xi: numpy array class

####

def HMM\_E\_step(vec\_Xi, mat\_A, vec\_lambda, vec\_pi) :

mat\_emission = get\_mat\_emission(vec\_Xi, vec\_lambda)

vec\_C, mat\_alpha = get\_alpha\_C(mat\_A, vec\_pi, mat\_emission)

mat\_beta = get\_beta(mat\_A, vec\_pi, mat\_emission, vec\_C)

mat\_Gamma, mat\_Xi = get\_Gamma\_Xi(mat\_A, mat\_emission, mat\_alpha, mat\_beta, vec\_C)

res = [mat\_Gamma, mat\_Xi]

return res

def get\_mat\_emission(vec\_Xi, vec\_lambda) :

mat\_emission = np.array([[pow(x, y) \* pow(math.e, -1.0 \* x) / math.factorial(y) for x in vec\_lambda] for y in vec\_Xi])

return mat\_emission

def get\_alpha\_C(mat\_A, vec\_pi, mat\_emission) :

num\_of\_states = len(vec\_pi)

num\_of\_obs = len(mat\_emission)

alpha\_0 = np.array([mat\_emission[0][i] \* vec\_pi[i] for i in range(0, num\_of\_states)])

C\_0 = 0.0

for alpha in alpha\_0 :

C\_0 += alpha

vec\_C\_buf = np.empty(num\_of\_obs)

vec\_C\_buf[0] = (C\_0)

alpha\_0 = alpha\_0 / C\_0

mat\_alpha\_buf = []

mat\_alpha\_buf.append(list(alpha\_0.copy()))

for n in range(1, num\_of\_obs) :

alpha\_n = np.empty([num\_of\_states])

for i in range(0, num\_of\_states) :

sum\_j = 0.0

for j in range(0, num\_of\_states) :

sum\_j += mat\_alpha\_buf[n - 1][j] \* mat\_A[j][i]

alpha\_n\_i = mat\_emission[n][i] \* sum\_j

alpha\_n[i] = alpha\_n\_i

C\_n = 0.0

for alpha in alpha\_n :

C\_n += alpha

vec\_C\_buf[n] = pd.Series([C\_n])

alpha\_n = alpha\_n / C\_n

mat\_alpha\_buf.append(list(alpha\_n.copy()))

res = [vec\_C\_buf, mat\_alpha\_buf]

return res

def get\_beta(mat\_A, vec\_pi, mat\_emission, vec\_C) :

num\_of\_states = len(vec\_pi)

num\_of\_obs = len(mat\_emission)

# initialization

mat\_beta\_buf = np.zeros([num\_of\_obs, num\_of\_states])

for i in range(0, num\_of\_states) :

mat\_beta\_buf[num\_of\_obs - 1][i] = 1.0

for m in range(1, num\_of\_obs) :

n = num\_of\_obs - 1 - m

mat\_beta\_buf\_n1 = mat\_beta\_buf[n + 1]

mat\_emission\_n1 = mat\_emission[n + 1]

mat\_beta\_buf\_n = mat\_beta\_buf[n]

vec\_C\_n1 = vec\_C[n + 1]

for i in range(0, num\_of\_states) :

sum\_j = 0.0

mat\_A\_i = mat\_A[i]

for j in range(0, num\_of\_states) :

sum\_j += mat\_beta\_buf\_n1[j] \* mat\_emission\_n1[j] \* mat\_A\_i[j]

mat\_beta\_buf\_n[i] = (sum\_j / vec\_C\_n1)

return mat\_beta\_buf

def get\_Gamma\_Xi(mat\_A, mat\_emission, mat\_alpha, mat\_beta, vec\_C) :

num\_of\_states = len(mat\_emission[0])

num\_of\_obs = len(mat\_emission)

mat\_Gamma\_buf = np.zeros([num\_of\_obs, num\_of\_states])

mat\_Gamma\_buf = mat\_alpha \* mat\_beta

mat\_Xi\_buf = np.zeros([num\_of\_obs - 1, num\_of\_states, num\_of\_states])

for m in range(0, num\_of\_obs - 1) :

mat\_Xi\_buf\_m = mat\_Xi\_buf[m]

mat\_alpha\_m = mat\_alpha[m]

mat\_emission\_m1 = mat\_emission[m + 1]

mat\_beta\_m1 = mat\_beta[m + 1]

vec\_C\_m1 = vec\_C[m + 1]

for i in range(0, num\_of\_states) :

mat\_Xi\_buf\_m\_i = mat\_Xi\_buf\_m[i]

mat\_alpha\_m\_i = mat\_alpha\_m[i]

mat\_A\_i = mat\_A[i]

for j in range(0, num\_of\_states) :

mat\_Xi\_buf\_m\_i[j] = (mat\_alpha\_m\_i \* mat\_emission\_m1[j] \* mat\_A\_i[j] \* mat\_beta\_m1[j]) / vec\_C\_m1

res = [mat\_Gamma\_buf, mat\_Xi\_buf]

return res

####

# HMM\_M\_step

# carries out the M step.

# updates (vector pi, vector lambda, matrix A).

# arguments:

# vec\_Xi: numpy array class

# mat\_A: numpy array class

# vec\_lambda: numpy array class

# vec\_pi: numpy array class

# mat\_Gamma: numpy array class

# mat\_Xi: numpy array class

# returns

# vec\_pi\_new: numpy array class

# vec\_lambda\_new: numpy array class

# mat\_A\_new: numpy array class

####

def HMM\_M\_step(vec\_Xi, mat\_A, vec\_lambda, vec\_pi, mat\_Gamma, mat\_Xi) :

num\_of\_states = len(mat\_A)

num\_of\_obs = len(vec\_Xi)

pi\_denom = 0.0

pi\_denom += sum(mat\_Gamma[0])

vec\_pi\_new = mat\_Gamma[0] / pi\_denom

# maxmize wrt lambda vector

vec\_lambda\_new = np.empty(num\_of\_states)

for k in range(0, num\_of\_states) :

lambda\_denom = 0.0

lambda\_nume = 0.0

for n in range(0, num\_of\_obs) :

lambda\_denom += mat\_Gamma[n][k]

lambda\_nume += mat\_Gamma[n][k] \* vec\_Xi[n]

if(lambda\_denom == 0.0) :

vec\_lambda\_new[k] = 0.0

else :

vec\_lambda\_new[k] = lambda\_nume / lambda\_denom

# maxmize wrt A matrix

mat\_A\_new = np.zeros([num\_of\_states, num\_of\_states])

for j in range(0, num\_of\_states) :

A\_denome = 0.0

for n in range(0, num\_of\_obs - 1) :

for mat\_Xi\_n\_j\_l in mat\_Xi[n][j] :

A\_denome += mat\_Xi\_n\_j\_l

for k in range(0, num\_of\_states) :

A\_nume = 0.0

for mat\_Xi\_n in mat\_Xi :

A\_nume += mat\_Xi\_n[j][k]

if(A\_denome == 0.0) :

mat\_A\_new[j][k] = 0.0

else :

mat\_A\_new[j][k] = A\_nume / A\_denome

res = [vec\_pi\_new, vec\_lambda\_new, mat\_A\_new]

return res

####

# HMM\_Viterbi

# carries out the Viterbi algorithm.

# arguments:

# vec\_Xi: numpy array class

# mat\_A: numpy array class

# vec\_lambda: numpy array class

# vec\_pi: numpy array class

# returns

# vec\_hs\_seq: index of the optimal vec\_lambda

####

def HMM\_Viterbi(vec\_Xi, mat\_A, vec\_lambda, vec\_pi) :

mat\_emission = get\_mat\_emission(vec\_Xi, vec\_lambda)

num\_of\_states = len(mat\_A)

num\_of\_obs = len(vec\_Xi)

mat\_hs\_seq = np.zeros([num\_of\_states, num\_of\_obs])

vec\_logp\_seq = np.zeros(num\_of\_states)

for j in range(0, num\_of\_states) :

mat\_hs\_seq[j][0] = j

if (vec\_pi[j] \* mat\_emission[0][j] == 0) :

vec\_logp\_seq[j] = -np.inf

else :

vec\_logp\_seq[j] = math.log(vec\_pi[j] \* mat\_emission[0][j]) / math.log(10)

for n in range(1, num\_of\_obs) :

mat\_hs\_seq\_buf = mat\_hs\_seq.copy()

vec\_logp\_seq\_buf = vec\_logp\_seq.copy()

for j in range(0, num\_of\_states) :

vec\_h\_logprob\_i = np.zeros(num\_of\_states)

for i in range(0, num\_of\_states) :

vec\_h\_logprob\_i[i] = vec\_logp\_seq[i] + math.log(mat\_emission[n][j] \* mat\_A[i][j]) / math.log(10)

max\_element = max(vec\_h\_logprob\_i)

max\_pos = np.where(vec\_h\_logprob\_i == max\_element)[0][0]

vec\_logp\_seq\_buf[j] = max\_element

mat\_hs\_seq\_buf[j] = mat\_hs\_seq[max\_pos].copy()

mat\_hs\_seq\_buf[j][n] = j

mat\_hs\_seq = mat\_hs\_seq\_buf.copy()

vec\_logp\_seq = vec\_logp\_seq\_buf.copy()

max\_element = max(vec\_logp\_seq)

max\_pos = np.where(vec\_logp\_seq == max\_element)[0][0]

vec\_hs\_seq = mat\_hs\_seq[max\_pos].copy()

return vec\_hs\_seq