##########

# 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

# Physical Review E (2014) 89:022705

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

# Contact:

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

##########

###########################################

# HMM\_v2.py

# 2017/11/13 revised by Daisuke Endo

# endo.daisuke.63m@st.kyoto-u.ac.jp

###########################################

#############

# Instruction

#

# put HMM\_v1.py in a folder on a path.

# import HMM\_v1 as HMM

# then executable as HMM.(function name)

#

# the function HMM takes a spike train as an argument

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

# read a text file of a spike train by data = np.loadtxt("data.txt")

# this program computes hidden variables given as instantaneous rate, using the HMM and draw a figure of the rate.

#

# you need libraries of matplotlib, numpy, and pandas

# 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 numpy as np

import math

def hmm(spike\_times):

"""

argument: spike\_times: given in list or ndarray

returns hidden states (rates) and draw the figure

observed values is not binary (0,1) but the number of spikes in a given time bin.

Poisson distribution is assumed for the number of spikes for a given time bin.

python HMM\_v1.py:

if \_\_name\_\_ == "\_\_main\_\_":

data = np.loadtxt("data.txt")

hmm(spike\_times=data)

# HMM.py:

import numpy as np

import matplotlib.pyplot as plt

import math

import HMM\_v1 as HMM

data = np.loadtxt("data.txt")

HMM.hmm(spike\_times=data)

"""

################################

# determine the times of initiation and termination

# bin size = 5\*(inter-spike interval)

##############################

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

###############################

# get\_hmm\_ratefunc: compute the transition between the hidden states

# rate\_hmm = (time, rate) given in a form of ndarray

# drawHMM: draw a figure

##############################

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

drawHMM(spike\_times, rate\_hmm)

return rate\_hmm

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

"""

draw a figure of the estimated state (given in a form of firing rate)

arguments:

spike\_times : given in list or ndarray

rate\_hmm = (time, rate) determined by the Baum-Welch algorithm and the Viterbi algorithm in a form of ndarray

returns:

nothing, but draws a figure

"""

###################

# output is shaped so that state changes appear vertical.

###################

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()

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

"""

infers optimal states using the Baum-Welch algorithm and the Viterbi algorithm

arguments:

spike\_times

bin\_width: bin size

max\_value: final spike time

min\_value: initial spike time

returns:

hidden states (rate\_func). Here rate\_func is given by a matrix of (time, state)

example:

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

"""

#######################

# set the initial values of the model parameters

# vec\_spkt: sets the initial spike time

# vec\_Xi: acquires the observed values

# vec\_Xi consists of the number of spikes (0, 1, 2, 3, ..) in each step.

#######################

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)

#########################################################

#

# Optimizing the model parameters using the Baum-Welch algorithm

#

# updates parameters by hmm\_E\_step and hmm\_M\_step

########################################################

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

mat\_A\_old = mat\_A

vec\_pi\_old = vec\_pi

vec\_lambda\_old = vec\_lambda

#####################

# Evaluation in the while loop

# stops when the change in a model parameter becomes small

# set flag=1 if the sum of change in a parameter sumcheck becomes smaller than some threshold

# or stops if the loops are repeated so many times

#####################

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

vec\_lambda = vec\_lambda\_new

mat\_A = mat\_A\_new

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

vec\_pi\_old = vec\_pi

vec\_lambda\_old = vec\_lambda

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

loop += 1

#############################################

#

# Estimate an optimal sequence of states using the Viterbi algorithm

# state is represented as 0 or 1 here

#

#############################################

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

############################################

# vec\_hidden: 0 or 1, representing the hidden state

# two states are transformed into the rates

############################################

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

################################

#

# Function acquiring the observation sequence from a spike train

#

################################

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

"""

arguments:

vec\_spkt: spike time measured from the initial spike

bin\_width: bin size

returns:

vec\_Xi: observation values consisting of spike counts in each bin.

"""

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

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

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

##########

# counting spikes

##########

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

# computes expectation

#

###########################

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

"""

arguments:

vec\_Xi: observation consisting of spike counts

mat\_A: transition matrix

vec\_lambda: spikes in each bin

vec\_pi: initial probabilities

returns:

mat\_Gamma: a matrix consisting of P(state i at time t, vec\_Xi |model)

mat\_Xi: a matrix consisting of P(state i at time t and state j at time t+1, vec\_Xi|model)

"""

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)

return mat\_Gamma, mat\_Xi

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

"""

arguments:

vec\_Xi: observation

vec\_lambda: average spikes in each bin

returns:

mat\_emission: probability of obtaining each observation in each state

mat\_emission: matrix consisting of (step,state)

"""

############

# assuming the Poisson distribution

# vec\_lambda: parameter of the Poisson distribution representing the mean

# gives the probability of having the observation

#############

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):

"""

arguments:

mat\_A: transition matrix

vec\_pi: initial probability

mat\_emission: matrix consisting of the probability of having the observation (step,state)

returns:

mat\_alpha: forward parameter

vec\_C(coefficient): scaling coefficient

"""

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

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

mat\_alpha = np.empty((num\_of\_obs, num\_of\_states))

coefficient = np.empty(num\_of\_obs)

#####################

# foward algorithm

#

# initialization

mat\_alpha[0, :] = vec\_pi \* mat\_emission[0, :]

# induction

for t in range(0, num\_of\_obs - 1): # do scaling

coefficient[t] = np.sum(mat\_alpha[t, :])

mat\_alpha[t, :] = mat\_alpha[t, :] / coefficient[t]

mat\_alpha[t+1, :] = (mat\_alpha[t, :] @ mat\_A) \* mat\_emission[t+1, :]

coefficient[num\_of\_obs - 1] = np.sum(mat\_alpha[num\_of\_obs - 1, :])

mat\_alpha[num\_of\_obs - 1, :] = mat\_alpha[num\_of\_obs - 1, :] / coefficient[num\_of\_obs - 1]

return coefficient, mat\_alpha

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

"""

arguments:

mat\_A: transition matrix

vec\_pi: initial probability

mat\_emission: matrix consisting of the probability of having the observation (step,state)

vec\_C: scaling coefficient obtained when computing alpha

returns:

mat\_beta: backward parameter

"""

##############################

# note given by Endo

# here we do not introduce matrix calculation for the state

# because the for-loop calculation was more rapid.

# we leave here the code for matrix computation.

##############################

##############################

# note:

# when computing mat\_beta[t, :], it is not divided by vec\_C[t]

# to avoid duplication when computing mat\_Gamma

# when computing mat\_Xi, it was divided by vec\_C[t+1] to make end meets. ##############################

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

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

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

###############

# backward algorithm

#

# initialization

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

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

# induction

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

n = num\_of\_obs - 1 - m

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

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

mat\_beta\_n = mat\_beta[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\_n1[j] \* mat\_emission\_n1[j] \* mat\_A\_i[j]

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

return mat\_beta

'''

# the following is the matrix computation

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

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

beta = np.empty((num\_of\_obs, num\_of\_states))

######################

# backward algorithm

# initialization beta

beta[num\_of\_obs - 1, :] = 1

# induction

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

beta[t - 1, :] = np.sum((mat\_A \* mat\_emission[t, :] \* beta[t, :]), axis=1) / vec\_C[t]

return beta'''

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

"""

arguments:

mat\_A: transition matrix

vec\_pi: initial probability

mat\_emission: matrix consisting of the probability of having the observation (step,state)

vec\_C: scaling coefficient obtained when computing alpha

returns:

mat\_Gamma: a matrix consisting of P(state i at time t, vec\_Xi |model)

mat\_Xi: a matrix consisting of P(state i at time t and state j at time t+1, vec\_Xi|model)

"""

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

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

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

mat\_Xi = np.empty((num\_of\_obs - 1, num\_of\_states, num\_of\_states))

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

mat\_Xi[t, :, :] = mat\_alpha[t, :].reshape(

num\_of\_states, 1) \* mat\_A \* mat\_emission[t + 1, :] \* mat\_beta[t + 1, :] / vec\_C[t+1]

################

# note

# procedure of dividing by vec\_C when computing mat\_Xi is complicated but

# this is because mat\_beta[t+1, :] was not divided by vec\_C[t+1].

################

return mat\_Gamma, mat\_Xi

###########################

#

# hmm\_M\_step

# this is a M step, maximizing the likelihood

#

###########################

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

"""

arguments:

mat\_A: transition matrix

vec\_pi: initial probability

mat\_Gamma: a matrix consisting of P(state i at time t, vec\_Xi |model)

mat\_Xi: a matrix consisting of P(state i at time t and state j at time t+1, vec\_Xi|model)

returns:

updated parameters: vec\_pi\_new, vec\_lambda\_new, mat\_A\_new

"""

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

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

#############################

#

# computes new parameters

#

#############################

#################

# vec\_pi

################

vec\_pi\_new = mat\_Gamma[0] / np.sum(mat\_Gamma[0])

#################

# vec\_lambda

#################

vec\_lambda\_new = np.sum(mat\_Gamma \* vec\_Xi.reshape(num\_of\_obs, 1), axis=0) / np.sum(mat\_Gamma, axis=0)

###############

# mat\_A

###############

total\_gamma\_a = np.zeros((1, num\_of\_states))

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

total\_gamma\_a += mat\_Gamma[t, :]

total\_xi = np.sum(mat\_Xi, axis=0) # (i,j): 2 by 2 matrix

mat\_A\_new = total\_xi / total\_gamma\_a.T # division between i-th ones

return vec\_pi\_new, vec\_lambda\_new, mat\_A\_new

#####################################

#

# a function for determining an optimal state sequence with the Viterbi algorithm

#

#####################################

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

"""

arguments:

vex\_Xi: observation sequence

mat\_A: transition matrix

vec\_lambda: average spike rate in each bin

vec\_pi: initial probability

returns:

vec\_hs\_seq: optimal state sequence

"""

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):

# copy the seq. up to n - 1

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