```
########################################
# CS/CNS/EE 155 2018
# Problem Set 6
#
# Author:       Andrew Kang
# Description:  Set 6 skeleton code
########################################

# You can use this (optional) skeleton code to complete the HMM
# implementation of set 5. Once each part is implemented, you can simply
# execute the related problem scripts (e.g. run 'python 2G.py') to quickly
# see the results from your code.
#
# Some pointers to get you started:
#
#     - Choose your notation carefully and consistently! Readable
#       notation will make all the difference in the time it takes you
#       to implement this class, as well as how difficult it is to debug.
#
#     - Read the documentation in this file! Make sure you know what
#       is expected from each function and what each variable is.
#
#     - Any reference to "the (i, j)^th" element of a matrix T means that
#       you should use T[i][j].
#
#     - Note that in our solution code, no NumPy was used. That is, there
#       are no fancy tricks here, just basic coding. If you understand HMMs
#       to a thorough extent, the rest of this implementation should come
#       naturally. However, if you'd like to use NumPy, feel free to.
#
#     - Take one step at a time! Move onto the next algorithm to implement
#       only if you're absolutely sure that all previous algorithms are
#       correct. We are providing you waypoints for this reason.
#
# To get started, just fill in code where indicated. Best of luck!

import random
import numpy as np

class HiddenMarkovModel:
    '''
    Class implementation of Hidden Markov Models.
    '''

    def __init__(self, A, O):
        '''
        Initializes an HMM. Assumes the following:
            - States and observations are integers starting from 0.
            - There is a start state (see notes on A_start below). There
              is no integer associated with the start state, only
              probabilities in the vector A_start.
```

```
        - There is no end state.

    Arguments:
        A:          Transition matrix with dimensions L x L.
                    The (i, j)^th element is the probability of
                    transitioning from state i to state j. Note that
                    this does not include the starting probabilities.

        O:          Observation matrix with dimensions L x D.
                    The (i, j)^th element is the probability of
                    emitting observation j given state i.

    Parameters:
        L:          Number of states.

        D:          Number of observations.

        A:          The transition matrix.

        O:          The observation matrix.

        A_start:    Starting transition probabilities. The i^th element
                    is the probability of transitioning from the start
                    state to state i. For simplicity, we assume that
                    this distribution is uniform.
    '''

    self.L = len(A)
    self.D = len(O[0])
    self.A = A
    self.O = O
    self.A_start = [1. / self.L for _ in range(self.L)]


def viterbi(self, x):
    '''
    Uses the Viterbi algorithm to find the max probability state
    sequence corresponding to a given input sequence.

    Arguments:
        x:          Input sequence in the form of a list of length M,
                    consisting of integers ranging from 0 to D - 1.

    Returns:
        max_seq:    State sequence corresponding to x with the highest
                    probability.
    '''

    M = len(x)      # Length of sequence.
    A = self.A
    O = self.O
```

```
    L = self.L
    A_start = self.A_start

    # using list comprehensions for more concise code
    # The (i, j)^th elements of probs and seqs are the max probability
    # of the prefix of length i ending in state j and the prefix
    # that gives this probability, respectively.
    #
    # For instance, probs[1][0] is the probability of the prefix of
    # length 1 ending in state 0.
    probs = [[0. for _ in range(self.L)] for _ in range(M + 1)]
    seqs = [['' for _ in range(self.L)] for _ in range(M + 1)]

    # initialization case
    for state in range(L):
        # log probability of transition into starting state + generating
         x[0]
        # probs[0][state] = (np.log(self.A_start[state]) +
         np.log(self.O[state][x[0]])).tolist()
        probs[0][state] = A_start[state] * O[state][x[0]]

        seqs[0][state] = str(state)

    # Viterbi for all rest of states
    # loop over all observations
    for prev_x in range(M-1):
        # loop over all states
        for state in range(L):
            k = prev_x
            y = state
            # (cur_prob, append_state) = max(((np.log(probs[k][y_prev]) +
             np.log(A[y_prev][y]) + np.log(O[y][x[k+1]])).tolist(),
             y_prev) for y_prev in range(L))
            (cur_prob, append_state) = max((probs[k][y_prev] *
             A[y_prev][y] * O[y][x[k+1]],y_prev) for y_prev in range(L))


            probs[k+1][y] = cur_prob
            seqs[k+1][y] = seqs[k][append_state] + str(y)

    k += 1
    (prob, state) = max((probs[M-1][state], y) for y in range(L))
    max_seq = seqs[M-1][state]
    return max_seq


def forward(self, x, normalize=False):
    '''
    Uses the forward algorithm to calculate the alpha probability
    vectors corresponding to a given input sequence.
```

```
    Arguments:
        x:          Input sequence in the form of a list of length M,
                    consisting of integers ranging from 0 to D - 1.

        normalize:  Whether to normalize each set of alpha_j(i) vectors
                    at each i. This is useful to avoid underflow in
                    unsupervised learning.

    Returns:
        alphas:     Vector of alphas.

                    The (i, j)^th element of alphas is alpha_j(i),
                    i.e. the probability of observing prefix x^1:i
                    and state y^i = j.

                    e.g. alphas[1][0] corresponds to the probability
                    of observing x^1:1, i.e. the first observation,
                    given that y^1 = 0, i.e. the first state is 0.
    '''
    M = len(x)      # Length of sequence.
    # Hmm, seems like have to use M+1 and not M
    alphas = [[1. for _ in range(self.L)] for _ in range(M+1)]


    # initialization here
    for state in range(self.L):
        alphas[1][state] = self.A_start[state] * self.O[state][x[0]]
        # if normalize:
        #     C = np.sum(alphas[1])
        #     alphas[1] = np.multiply((1/C), alphas[1]).tolist()

    if normalize:
        C = np.sum(alphas[1])
        alphas[1] = np.multiply((1/C), alphas[1]).tolist()

    # have normalization option for underflow issues.  keep running algo


    for k in range(1, M):
            # for state
            for y in range(self.L):
                # similar to Viterbi, but summing instead of max.
                 actually identical, obviously don't return the seq.
                alpha = sum(alphas[k][y_prev] * self.A[y_prev][y] *
                 self.O[y][x[k]] for y_prev in range(self.L))
                alphas[k+1][y] = alpha
            if normalize:
                C = np.sum(alphas[k+1])
                alphas[k+1] = np.multiply((1/C), alphas[k+1]).tolist()

    return alphas
```

```
def backward(self, x, normalize=False):
    '''
    Uses the backward algorithm to calculate the beta probability
    vectors corresponding to a given input sequence.

    Arguments:
        x:          Input sequence in the form of a list of length M,
                    consisting of integers ranging from 0 to D - 1.

        normalize:  Whether to normalize each set of alpha_j(i) vectors
                    at each i. This is useful to avoid underflow in
                    unsupervised learning.

    Returns:
        betas:      Vector of betas.

                    The (i, j)^th element of betas is beta_j(i), i.e.
                    the probability of observing prefix x^(i+1):M and
                    state y^i = j.

                    e.g. betas[M][0] corresponds to the probability
                    of observing x^M+1:M, i.e. no observations,
                    given that y^M = 0, i.e. the last state is 0.
    '''

    M = len(x)      # Length of sequence.
    betas = [[0. for _ in range(self.L)] for _ in range(M + 1)]

    # initialization here
    for state in range(self.L):
        betas[M][state] = 1.0
        # if normalize:
        #     C = np.sum(betas[M])
        #     betas[1] = np.multiply((1/C), betas[M]).tolist()

    if normalize:
        C = np.sum(betas[M])
        betas[1] = np.multiply((1/C), betas[M]).tolist()

    # have normalization option for underflow issues.  keep running algo

    # This time we are running backwards, so start at M, go to 0
    for k in range(M, 0, -1):
            # for state
            for y in range(self.L):
                # similar to Viterbi, but summing instead of max.
                 actually identical, obviously don't return the seq.
                beta = sum(betas[k][y_next] * self.A[y][y_next] *
                 self.O[y_next][x[k-1]] for y_next in range(self.L))
```

```
                betas[k-1][y] = beta
            if normalize:
                C = np.sum(betas[k-1])
                betas[k-1] = np.multiply((1/C), betas[k-1]).tolist()


    return betas


def supervised_learning(self, X, Y):
    '''
    Trains the HMM using the Maximum Likelihood closed form solutions
    for the transition and observation matrices on a labeled
    datset (X, Y). Note that this method does not return anything, but
    instead updates the attributes of the HMM object.

    Arguments:
        X:          A dataset consisting of input sequences in the form
                    of lists of variable length, consisting of integers
                    ranging from 0 to D - 1. In other words, a list of
                    lists.

        Y:          A dataset consisting of state sequences in the form
                    of lists of variable length, consisting of integers
                    ranging from 0 to L - 1. In other words, a list of
                    lists.

                    Note that the elements in X line up with those in Y.
    '''

    # Calculate each element of A using the M-step formulas.
    # Determine number of sequences
    N = len(X)
    # A: transition matrix
    for a in range(self.L):
        for b in range(self.L):
            a_numer = 0
            a_denom = 0

            for j in range(N):
                M = len(X[j])
                for i in range(M-1):
                    if (Y[j][i] == a) & (Y[j][i+1] == b):
                        a_numer += 1
                    if (Y[j][i] == a):
                        a_denom += 1
            self.A[a][b] = a_numer / a_denom



    # Calculate each element of O using the M-step formulas.
```

```
    for z in range(self.L):
        for w in range(self.D):
            o_numer = 0
            o_denom = 0

            for j in range(N):
                M = len(X[j])
                for i in range(M):
                    if (Y[j][i] == z) & (X[j][i] == w):
                        o_numer += 1
                    if (Y[j][i] == z):
                        o_denom += 1
            self.O[z][w] = o_numer / o_denom

    pass


def unsupervised_learning(self, X, N_iters):
    '''
    Trains the HMM using the Baum-Welch algorithm on an unlabeled
    datset X. Note that this method does not return anything, but
    instead updates the attributes of the HMM object.

    Arguments:
        X:          A dataset consisting of input sequences in the form
                    of lists of length M, consisting of integers ranging
                    from 0 to D - 1. In other words, a list of lists.

        N_iters:    The number of iterations to train on.
    '''

    for each_iteration in range(N_iters):
        # Primarily sticking to list comprehensions for more readable code
        # initialize the 0 matrices for A numerator and denominator and O
         num/denom
        a_num = [[0. for _ in range(self.L)] for _ in range(self.L)]
        a_den = [[0. for _ in range(self.L)] for _ in range(self.L)]
        o_num = [[0. for _ in range(self.D)] for _ in range(self.L)]
        o_den = [[0. for _ in range(self.D)] for _ in range(self.L)]

        for j in range(len(X)):

            M = len(X[j])


            # initialize the probabilities.
            probs = [[0. for _ in range(self.L)] for _ in range(M)]
            joint_probs = [[[0. for _ in range(self.L)] for _ in
             range(self.L)] for _ in range(M)]
```

```
        # time for E/M
        # E:
        # Calculate alphas and betas using forward and backward methods
        alphas = self.forward(X[j], normalize=True)
        betas = self.backward(X[j], normalize=True)

        # probabilities now
        for i in range(M):
            # marginal
            probs[i] = [alphas[i+1][z] * betas[i+1][z] /
             sum([alphas[i+1][w] * betas[i+1][w] for w in
             range(self.L)]) for z in range(self.L)]

        for i in range(M-1):
            joint_margin_denom = sum([sum([alphas[i+1][a] *
             betas[i+1+1][b] * self.A[a][b] * self.O[b][X[j][i+1]]
                for a in range(self.L)]) for b in range(self.L)])

            for a in range(self.L):
                for b in range(self.L):
                    cur_joint_prob = alphas[i+1][a] * betas[i+1+1][b]
                     * self.A[a][b] * self.O[b][X[j][i+1]]
                    joint_probs[i][a][b] = cur_joint_prob /
                     joint_margin_denom

        # so many current iterator variables in use, be very careful.
         removing all cases of ii, jj, i_, etc and replacing with
        # single letters
        # M:
        # update A
        for a in range(self.L):
            for b in range(self.L):
                a_num[a][b] += sum([joint_probs[i][a][b] for i in
                 range(M-1)])
                a_den[a][b] += sum([probs[i][a] for i in range(M-1)])
        # update O
        # over states
        for z in range(self.L):
            # over tokens
            for w in range(self.D):
                o_den[z][w] += sum([probs[i][z] for i in range(M)])
                for i in range(M):
                    if X[j][i] ==w:
                        o_num[z][w] += probs[i][z]

    self.A = [[a_num[i][j] /a_den[i][j] for j in range(self.L)] for i
     in range(self.L)]
    self.O = [[o_num[i][j] / o_den[i][j] for j in range(self.D)] for i
     in range(self.L)]
```

```
def generate_emission(self, M):
    '''
    Generates an emission of length M, assuming that the starting state
    is chosen uniformly at random.

    Arguments:
        M:          Length of the emission to generate.

    Returns:
        emission:   The randomly generated emission as a list.

        states:     The randomly generated states as a list.
    '''

    emission = []
    states = []

    # start state randomly
    states.append(np.random.choice(self.L))
    emission.append(np.random.choice(self.D, p=self.O[states[0]]))
    # now loop over
    for i in range(M-1):
        states.append(np.random.choice(self.L, p=self.A[states[i]]))
        emission.append(np.random.choice(self.D, p=self.O[states[i]]))


    return emission, states


def probability_alphas(self, x):
    '''
    Finds the maximum probability of a given input sequence using
    the forward algorithm.

    Arguments:
        x:          Input sequence in the form of a list of length M,
                    consisting of integers ranging from 0 to D - 1.

    Returns:
        prob:       Total probability that x can occur.
    '''

    # Calculate alpha vectors.
    alphas = self.forward(x)

    # alpha_j(M) gives the probability that the state sequence ends
    # in j. Summing this value over all possible states j gives the
```

```
        # total probability of x paired with any state sequence, i.e.
        # the probability of x.
        prob = sum(alphas[-1])
        return prob


    def probability_betas(self, x):
        '''
        Finds the maximum probability of a given input sequence using
        the backward algorithm.

        Arguments:
            x:          Input sequence in the form of a list of length M,
                        consisting of integers ranging from 0 to D - 1.

        Returns:
            prob:       Total probability that x can occur.
        '''

        betas = self.backward(x)

        # beta_j(1) gives the probability that the state sequence starts
        # with j. Summing this, multiplied by the starting transition
        # probability and the observation probability, over all states
        # gives the total probability of x paired with any state
        # sequence, i.e. the probability of x.
        prob = sum([betas[1][j] * self.A_start[j] * self.O[j][x[0]] \
                    for j in range(self.L)])

        return prob


def supervised_HMM(X, Y):
    '''
    Helper function to train a supervised HMM. The function determines the
    number of unique states and observations in the given data, initializes
    the transition and observation matrices, creates the HMM, and then runs
    the training function for supervised learning.

    Arguments:
        X:          A dataset consisting of input sequences in the form
                    of lists of variable length, consisting of integers
                    ranging from 0 to D - 1. In other words, a list of lists.

        Y:          A dataset consisting of state sequences in the form
                    of lists of variable length, consisting of integers
                    ranging from 0 to L - 1. In other words, a list of lists.
                    Note that the elements in X line up with those in Y.
    '''
    # Make a set of observations.
    observations = set()
```

```
    for x in X:
        observations |= set(x)

    # Make a set of states.
    states = set()
    for y in Y:
        states |= set(y)

    # Compute L and D.
    L = len(states)
    D = len(observations)

    # Randomly initialize and normalize matrix A.
    A = [[random.random() for i in range(L)] for j in range(L)]

    for i in range(len(A)):
        norm = sum(A[i])
        for j in range(len(A[i])):
            A[i][j] /= norm

    # Randomly initialize and normalize matrix O.
    O = [[random.random() for i in range(D)] for j in range(L)]

    for i in range(len(O)):
        norm = sum(O[i])
        for j in range(len(O[i])):
            O[i][j] /= norm

    # Train an HMM with labeled data.
    HMM = HiddenMarkovModel(A, O)
    HMM.supervised_learning(X, Y)

    return HMM

def unsupervised_HMM(X, n_states, N_iters):
    '''
    Helper function to train an unsupervised HMM. The function determines the
    number of unique observations in the given data, initializes
    the transition and observation matrices, creates the HMM, and then runs
    the training function for unsupervised learing.

    Arguments:
        X:          A dataset consisting of input sequences in the form
                    of lists of variable length, consisting of integers
                    ranging from 0 to D - 1. In other words, a list of lists.

        n_states:   Number of hidden states to use in training.

        N_iters:    The number of iterations to train on.
    '''
```

```
# Make a set of observations.
observations = set()
for x in X:
    observations |= set(x)

# Compute L and D.
L = n_states
D = len(observations)
random.seed(2020)
# Randomly initialize and normalize matrix A.
A = [[random.random() for i in range(L)] for j in range(L)]

for i in range(len(A)):
    norm = sum(A[i])
    for j in range(len(A[i])):
        A[i][j] /= norm
random.seed(155)
# Randomly initialize and normalize matrix O.
O = [[random.random() for i in range(D)] for j in range(L)]

for i in range(len(O)):
    norm = sum(O[i])
    for j in range(len(O[i])):
        O[i][j] /= norm

# Train an HMM with unlabeled data.
HMM = HiddenMarkovModel(A, O)
HMM.unsupervised_learning(X, N_iters)

return HMM
```