Hidden Markov Models (100 points)

In this programming assignment, we will implement Hidden Markov Models (HMM) and apply HMM to Part-of-Speech Tagging problem and Gene-tagging problem.

You need to first implement 6 important functions that help us accomplish various steps involved in learning HMM. Then you will be calculating HMM parameters from the data and use it to solve Part-of-Speech Tagging problem and Gene-tagging problem.

After finishing the implementation, you can use hmm_test_script.py to test the correctness of your functions.

Assignment must be submitted by **August 08, 23:59:59 PDT**

Office Hours

Date	Time	Meeting Link	CP
27th July [Monday]	5:00 PM to 7:00 PM	https://usc.zoom.us/j/5288651362	Sowmya
28th July [Tuesday]	10:30 AM to 12:30 PM	https://usc.zoom.us/j/6625239116	Anamay
30th July [Thursday]	10:00 AM - 12:00 PM	https://usc.zoom.us/j/98558703974	Amulya
3rd August [Monday]	5:00 PM to 7:00 PM	https://usc.zoom.us/j/5288651362	Sowmya
4th August [Tuesday]	10:30 AM to 12:30 PM	https://usc.zoom.us/j/6625239116	Anamay
6th August [Thursday]	10:00 AM - 12:00 PM	https://usc.zoom.us/j/98558703974	Amulya

Grading guideline

1.1 Implementation

1. `forward` function - 10 = 5x2 points
2. `backward` function - 10 = 5x2 points
3. `sequence_prob` function - 5 = 5x1 points

4. `posterior_prob` function - 10 = 5x2 points
5. `likelihood_prob` function - 10 = 5x2 points
6. `viterbi` function - 15 = 5*3 points

There are 5 sets of grading data. Each grading data includes paramaters `(pi, A, B, obs_dict, state_dict, Osequence)` , which we will use to initialize HMM class and test your functions. To receive full credits, your output of function 1-5 should within an error of 10^{-8}, your output of viterbi function should be identical with ours.

For the definitions of the parameters `pi` , `A` , `B` etc., please refer to the code documentation.

1.2 Application to Part-of-Speech Tagging

1. `model_training` function - 10 = 10x(your_correct_pred_cnt/our_correct_pred_cnt)
2. `sentence_tagging` function - 10 = 10x(your_correct_pred_cnt/our_correct_pred_cnt)

We will use the dataset given to you for grading this part (with a different random seed). We will train your model and our model on same train_data. `model_training` function and `sentence_tagging` function will be tested seperately.

In order to check your model_training function, we will use 50 sentences from `train_data` to do Part-of-Speech Tagging. To receive full credits, your prediction accuracy should be identical or better than ours.

In order to check your `sentence_tagging` function, we will use 50 sentences from `test_data` to do Part-of-Speech Tagging. To receive full credits, your prediction accuracy should be identical or better than ours.

1.3 Application to Gene Tagging

1. `model_training` function - 10 = 10*(your_correct_pred_cnt/our_correct_pred_cnt)
2. `sentence_tagging` function - 10 = 10*(your_correct_pred_cnt/our_correct_pred_cnt)

Evaluated similar to the previous application.

What to submit

- hmm.py
- tagger.py

1.1 Implementation (60 points)

In 1.1, you are given parameters of a HMM and you will implement two procedures.

1. **The Evaluation Problem** : Given HMM Model and a sequence of observations, what is the probability that the observations are generated by the model?

   Two algorithms are usually used for the evaluation problem: forward algorithm or the backward algorithm. Based on the result of forward algorithm and backward algorithm, you will be asked to calculate probability of sequence and posterior probability of state.

2. **The Decoding Problem** : Given a model and a sequence of observations, what is the most likely state sequence in the model which produced the observation sequence. For decoding you will be implementing Viterbi algorithm.

HMM Class

In this project, we abstracted Hidden Markov Model as a class. Each Hiddern Markov Model initialized with `Pi, A, B, obs_dict` and `state_dict` . HMM class has 6 inner functions: `forward` function, `backward` function, `sequence_prob` function, `posterior_prob` function, `likelihood_prob` and `viterbi` function.

```
###
You can add your own

function or variables in HMM class, but you shouldn 't change current existing api. ###
class HMM:
    def __init__(self, pi, A, B, obs_dict, state_dict):
    -pi: (1 * num_state) A numpy array of initial probailities.pi[i] = P(X_1 = s_i) -
    A: (num_state * num_state) A numpy array of transition probailities.A[i, j] = P(X_t = s
    B: (num_state * num_obs_symbol) A numpy array of observation probabilities.B[i, o] = P(
    obs_dict: A dictionary mapping each observation symbol to their index in B -
    state_dict: A dictionary mapping each state to their index in pi and A# TODO:
    def forward(self, Osequence): #TODO:
    def backward(self, Osequence): #TODO:
    def sequence_prob(self, Osequence): #TODO:
    def posterior_prob(self, Osequence): #TODO:
    def likelihood_prob(self, Osequence): #TODO:
    def viterbi(self, Osequence):
```

1.1.1 Evaluation problem

(a) Forward algorithm and backward algorithm (20 points)

Here λ means the model. Please finish the implementation of `forward()` function and `backward()` function in hmm.py:

- $\alpha[i, t] = P\left(X_t = s_i, Z_{1:t} | \lambda\right).$

```
def forward(self, Osequence):
    ""
"
Inputs:
    -self.pi: (1 * num_state) A numpy array of initial probailities.pi[i] = P(X_1 = s_i) -
    self.A: (num_state * num_state) A numpy array of transition probailities.A[i, j] = P(X_
    self.B: (num_state * num_obs_symbol) A numpy array of observation probabilities.B[i, o]
    Osequence: (1 * L) A numpy array of observation sequence with length L

Returns:
    -alpha: (num_state * L) A numpy array alpha[i, t] = P(X_t = s_i, Z_1: Z_t | λ)
""
"
```

- $\beta[i,t] = P\left(Z_{t+1:T} | X_t = s_i, \lambda\right).$

```
def backward(self, Osequence):
    ""
"
Inputs:
    -self.pi: (1 * num_state) A numpy array of initial probailities.pi[i] = P(X_1 = s_i) -
    self.A: (num_state * num_state) A numpy array of transition probailities.A[i, j] = P(X_
    self.B: (num_state * num_obs_symbol) A numpy array of observation probabilities.B[i, o]
    Osequence: (1 * L) A numpy array of observation sequence with length L

Returns:
    -beta: (num_state * L) A numpy array beta[i, t] = P(Z_t + 1: Z_T | X_t = s_i, λ)
""
"
```

(b) Sequence probability (5 points)

Based on your forward and backward function, you will calculate the sequence probability. (You can call forward function or backward function inside of `sequence_prob` function)

$$P\left(Z_1, ..., Z_T = O|\lambda\right) = \sum_{i=1}^{N} P\left(X_t = s_i, Z_{1:T}|\lambda\right) = \sum_{i=1}^{N} \alpha[i,T]$$

```
def sequence_prob(self, Osequence):
    ""
"
Inputs:
    -Osequence: (1 * L) A numpy array of observation sequence with length L

Returns:
```

```
    -prob: A float number of P(Z_1: Z_T | λ)
"""
"
```

© Posterior probability (10 points)

The forward variable $\alpha[i,t]$ and backward variable $\beta[i,t]$ are used to calculate the posterior probability of a specific state. Now for $t = 1 \dots T$ and $i = 1 \dots N$, we define posterior probability $\gamma_t(i) = P(X_t = s_i|O,\lambda)$ the probability of being in state s_i at time t given the observation sequence O and the model λ.

$$\gamma_t(i) = \frac{P(X_t = s_i, O|\lambda)}{P(O|\lambda)} = \frac{P(X_t = s_i, Z_{1:t}|\lambda)}{P(O|\lambda)}$$

$$P(X_t = s_i, Z_{1:t}|\lambda) = P(Z_{1:t}|X_t = s_i, \lambda) \cdot P(Z_{t+1:T}|X_t = s_i, \lambda) \cdot P(X_t = s_i|\lambda) = \alpha[i,t] \cdot \beta[i,t]$$

Thus

$$\gamma_t(i) = \frac{\alpha[i,t] \cdot \beta[i,t]}{P(O|\lambda)}$$

where

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha[i,T]$$

Signature:

```
def posterior_prob(self, Osequence):
    """
"
Inputs:
    -Osequence: (1 * L) A numpy array of observation sequence with length L

Returns:
    -prob: (num_state * L) A numpy array of P(X_t = i | O, λ)
"""
"
```

You can use $\beta_t(i)$ to find the most likely state at time t which is the state $Z_t = s_i$ for which $\beta_t(i)$ is maximum. This algorithm works fine in the case when HMM is ergodic i.e. there is transition from any state to any other state. If applied to an HMM of another architecture, this approach could give a sequence that may not be a legitimate path because some transitions are not permitted. To avoid this problem Viterbi algorithm is the most common decoding algorithms used.

(d) Likelihood of two consecutive states at a given time (10 points)

You are required to calculate the likelihood of transition from state s at time t to state s' at time $t+1$. That is, you're required to calculate

$$\xi_{s,s'}(t) = P(X_t = s, X_{t+1} = s' \mid Z_{1:T} = z_{1:T})$$

Signature:

```
def likelihood_prob(self, Osequence):
    ""
"
Inputs:
    -Osequence: (1 * L) A numpy array of observation sequence with length L

Returns:
    -prob: (num_state * num_state * (L - 1)) A numpy array of P(X_t = i, X_t + 1 = j | O, λ
""
"
```

(e) Viterbi algorithm (15 points)

Viterbi algorithm is a dynamic programming algorithm for finding the most likely sequence of hidden states. We want to compute the most likely state path that corresponds to the observation sequence O based HMM. Namely, $k^* = (k_1^*, k_2^*, \cdots, k_T^*) = \arg\max_k P(s_{k_1}, s_{k_2}, \cdots, s_{k_T} | Z_1, Z_2, \cdots, Z_T = O, \lambda)$.

Signature:

```
def viterbi(self, Osequence):
    ""
"
Inputs:
    -Osequence: (1 * L) A numpy array of observation sequence with length L

Returns:
    -path: A List of the most likely hidden state path k * (
        return state instead of idx)
```

" "
"

1.2 Application to Speech Tagging (20 points)

Part-of-Speech (POS) is a category of words (or, more generally, of lexical items) which have similar grammatical properties. (Example: noun, verb, adjective, adverb, pronoun, preposition, conjunction, interjection, and sometimes numeral, article, or determiner.)

Part-of-Speech Tagging (POST) is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context.

Here you will use HMM to do POST. You will need to calculate the parameters `(pi, A, B, obs_dict, state_dict)` of HMM first and then apply Viterbi algorithm to do speech-tagging.

Dataset

tags.txt: Universal Part-of-Speech Tagset

Tag	Meaning	English Examples
ADJ	adjective	new, good, high, special, big, local
ADP	adposition	on, of, at, with, by, into, under
ADV	adverb	really, already, still, early, now
CONJ	conjunction	and, or, but, if, while, although
DET	determiner, article	the, a, some, most, every, no, which
NOUN	noun	year, home, costs, time, Africa
NUM	numeral	twenty-four, fourth, 1991, 14:24
PRT	particle	at, on, out, over per, that, up, with
PRON	pronoun	he, their, her, its, my, I, us
VERB	verb	is, say, told, given, playing, would
.	punctuation marks	. , ; !
X	other	ersatz, esprit, dunno, gr8, univeristy

sentences.txt: Includes nearly 50000 sentences which have already been tagged.

Word	Tag
b100-48585	
She	PRON
had	VERB
to	PRT
move	VERB
in	ADP
some	DET
direction	NOUN
–	.
any	DET
direction	NOUN
that	PRON
would	VERB
take	VERB
her	PRON
away	ADV
from	ADP
this	DET
evil	ADJ
place	NOUN
.	.

Part-of-Speech Tagging

In this part, we collect our dataset and tags with Dataset class. Dataset class includes tags, train_data and test_data. In both dataset include a list of sentences, each sentence is an object of Line class.

You only need to implement `model_training` function and `sentence_tagging` function. We have provided the accuracy function, which you can use to compare your `predict_tagging` and `true_tagging` of a sentence. You can find the definition below.

```
###
You can add your own functions or variables in Dataset class, but you shouldn 't change cur
class Dataset:
    def __init__(self, tagfile, datafile, train_test_split = 0.8, seed = 112890):
    self.tags
self.train_data
self.test_data
def read_data(self, filename):
    def read_tags(self, filename):
    class Line:
    def __init__(self, line, type):
    self.id
self.words
self.tags
self.length
def show(self): #TODO:
    def model_training(train_data, tags)# TODO:
    def sentence_tagging(model, test_data, tags)
def accuracy(predict_tagging, true_tagging)
```

1.2.1 Model training (10 points)

In this part, you will need to calculate the parameters of HMM model based on `train_data` .

Signature:

```
def model_training(train_data, tags):
    ""
"
Inputs:
    -train_data: a list of sentences, each sentence is an object of Line class -
    tags: a list of POS tags

Returns:
    -model: an object of HMM class initialized with paramaters(pi, A, B, obs_dict, state_di
    ""
    "
```

1.2.2 Speech_tagging (10 points)

Based on HMM from 1.2.1, do speech tagging for each sentence on test data. Note when you meet a word which is unseen in training dataset. You need to modify the emission matrix and obs_dict of your current model in order to handle this case. You will assume the emission probability from each state to a new unseen word is 10^{-6}(a very low probability).

```
For example, in hmm_model.json, we use the following paramaters to initialize HMM:
    S = ["1", "2"]
pi: [0.7, 0.3]
A: [
    [0.8, 0.2],
    [0.4, 0.6]
]
B = [
    [0.5, 0, 0.4, 0.1],
    [0.5, 0.1, 0.2, 0.2]
]
Observations = ["A", "C", "G", "T"]
If we find another observation symbol "X" in observation sequence, we will modify parameter
    S = ["1", "2"]
pi: [0.7, 0.3]
A: [
    [0.8, 0.2],
    [0.4, 0.6]
]
B = [
    [0.5, 0, 0.4, 0.1, 1e-6],
    [0.5, 0.1, 0.2, 0.2, 1e-6]
]
Observations = ["A", "C", "G", "T", "X"]
```

You do not get access to `test_data` on `model_training` function, you need to implement the logic to tag a new sequence in `sentence_tagging` function.

Signature:

```
def sentence_tagging(test_data, model):
    ""
"
Inputs:
    -test_data: (1 * num_sentence) a list of sentences, each sentence is an object of Line
    model: an object of HMM class

Returns:
    -tagging: (num_sentence * num_tagging) a 2 D list of output tagging
```

```
for each sentences on test_data
    ""
"
```

1.2.3 Suggestion(0 points)

This part won't be graded. In order to have a better understanding of HMM. Come up with one sentence by yourself and tagging it manually. Then run your forward function, backward function, `seq_prob` function, `posterior_prob` function and viterbi function on the model from 1.2.1. Print the result of each function, see if you can explain your result.

1.3 Application to Gene Tagging (20 points)

In this task you'll use the HMM class to perform Gene tagging in sentences.
This application is similar to the previous POS Tagging application, the difference is that you'll tag gene names in biological text data. You will use use an HMM for this task. You will need to calculate the parameters `(pi, A, B, obs_dict, state_dict)` of HMM first and then apply Viterbi algorithm to do gene-tagging.

Dataset

gene_tags.txt: Tagset for the given gene dataset

Tag	Meaning	Examples
GENE	Gene names	5-neucliotidase, carbonic anhydrase etc.
O	Other	Any word that's not a gene

genes.txt: Includes about 3000 sentences which have already been tagged.

Word	Tag
Serum	GENE
gamma	GENE
glutamyltransferase	GENE
in	O

Word	Tag
the	O
diagnosis	O
of	O
liver	O
disease	O
in	O
cattle	O
.	O

Likelihood values

Use the likelihood values of consecutive states computed for a sentence to understand how the model is able to recognize gene names split into multiple words.

Note

Do not be deceived by the high tagging accuracy in this task. While our model can recognize the genes in most of the cases, even a model that tags every word as O would have a high accuracy since it's much more likely to find an O than a GENE. This is one of the classic examples where accuracy is not the right way to evaluate the model. Try some examples from the dataset to see this for yourself.