Ananya Kotha
7427344242

Python version: 3.10.12
Packages used:

```
import numpy as np
import json
```

For other instructions, please check the README file.

TASK 1

Training and development data are a list of dictionaries, each containing an 'index,' 'sentence' (list of words - already tokenized), and 'labels' corresponding to each word. The training data is read and stored in a list.

To find the 'vocab' for the words of the training dataset, I create a separate list and a corresponding dictionary called word_freq (in this case) stores the frequencies of each word. Also, I created a list called 'train_labels' to store all the unique training labels. (45 tags)

I chose a **threshold of 2**. So any word with a **frequency < 2** is considered an '< unk >', i.e. unknown tagged. I replaced all the words in train and dev datasets having a frequency less than threshold 2 as < unk >.

Then, the dictionary is sorted in descending order based on the occurrences, excluding the frequency of the < unk > tags. The sorted dictionary with < unk > as the first tag and the rest in descending order of frequency are written into a **'vocab.txt'** file.

What is the overall size of your vocabulary, and how many times does the special token "< unk >" occur following the replacement process?
Vocab size: 23183 (including < unk >)
Frequency of < unk > in train dataset: 20011

TASK 2

I calculated the initial, transition, and emission probabilities by iterating over the lines in the training data. First, I find the count of all possible states in initial states by iterating over the first word of each sentence in the training data. For transmission and emission probabilities, I iterate over the possible combinations of (prev_state, state) and (state, word) and keep track of counts. Here, I ignore the pairs which have a count of 0.

I created three new dictionaries to keep track of the above counts.
transition_counts - stores tuple of (prev_state, state) as key and number of times this combination occurs in training set as the value
emission_counts - stores tuple of (word, state) as key and number of times this combination occurs in training set as value

initial_state_counts = stores the state as key and the number of times this state occurs at the beginning of the sentence as value
overall_state_counts = stores the state as key and frequency of the state in the training data as value

Now, we can calculate the initial, emission, and transition probabilities and store them in the following dictionaries.

initial_probabilities - stores initial probabilities, i.e., when the given state is the first tag in the sentence, so has no prior tag to depend on
emission_probabilities: Stores the emission probabilities.
transition_probabilities: Stores the transition probabilities.

Initial_prob = initial_state_counts/total_no_of_sentences
Transition_prob = transition_counts/overall_state_counts of that state
Emission_prob = emission_counts/overall_state_counts of that state

The dictionaries initial_probabilities{}, transition_probability{}, and emission_probability{} are written into a **'hmm.json'** file, which is a dictionary having the above dictionaries as elements.

How many transition and emission parameters are in your HMM?
Number of Transition Parameters: 1351
Number of Emission Parameters: 30303

TASK 3

Using the algorithm mentioned,

First, I focus on determining the most probable part-of-speech tag (stored in most_probable_s) for the first word in each sentence. The code iterates through a list of possible part-of-speech tags (train_labels) and calculates the initial probability of each tag for the first word in the sentence. This probability is computed by multiplying the initial probability of the tag (i_prob.get(s, 1e-6)) with the emission probability of the tag for the first word in the sentence (e_prob.get((s, sentence[0]), 1e-6)). If a tag is not found in the probabilities, default small values (1e-6) are used. The code keeps track of the most probable tag (most_probable_s) by comparing the calculated probability (ini_prob_of_s) of each tag with the current maximum probability (final_prob_of_s). If a higher probability is found, it updates final_prob_of_s and most_probable_s accordingly.

Now, for words beyond the first word in the sentence (i.e., i > 0), I calculate the most probable part-of-speech tag given the previous tag (prev_tag). I initialized final_prob_of_s to negative infinity, tracking the maximum probability encountered. Iterating through all possible pairs of part-of-speech tags (train_labels) to calculate the probability of the current tag (s) being the correct tag for the current word in the sentence.

The probability is calculated as the product of two probabilities:

Transition probability (t_prob.get((prev_tag, s), 1e-6)) from the previous tag (prev_tag) to the current tag (s).
Emission probability (e_prob.get((s, sentence[i]), 1e-6)) of the current tag (s) emitting the current word in the sentence.
For each possible tag, the algorithm compares the calculated probability (prob_of_s) with the current maximum probability (final_prob_of_s). If the calculated probability is greater, it updates final_prob_of_s and most_probable_s to reflect the most likely tag encountered.

In this part of the code, I determine the most probable part-of-speech tag for words in the sentence (other than the first word) by considering both transition and emission probabilities. The algorithm selects the tag with the highest probability as the predicted tag for the current word in the sentence. This process is repeated for each word in the sentence as part of the greedy decoding algorithm.

Then I calculated accuracy by using accuracy = correctly_predicted_labels/total_labels

What is the accuracy of the dev data?
Accuracy on dev data after replacing the unknown words in the sentences with < unk > tags before prediction and using a smoothing of 1e-6 for all the probabilities is **0.93479.**

I created a function 'greedy_on_test_data' to create predictions for test.json and stored the predicted labels along with the sentence and index information in the file **'greedy.json.'**

TASK 4

The code uses dynamic programming to compute state probabilities for each position in a sentence. It iterates through possible part-of-speech tags (states) in a sentence. It starts by iterating through possible tags (S), computing the initial probabilities of each tag at the beginning of the sentence. Then, it iterates through each word (O) in the sentence, calculating the maximum probability of reaching a particular tag (S[s]) at each position. To achieve this, it considers all possible previous tags (S[k]). It calculates the probability of transitioning from S[k] to S[s], combined with the probability of the current word having the tag S[s]. The code updates the trellis matrix with these maximum probabilities and maintains the pointers matrix to record the previous state indices that lead to the highest probabilities.

For the final step of the Viterbi decoding process, the code determines the optimal sequence of part-of-speech tags (POS tags) for a given input sentence. It begins by initializing an empty list called best_path to store the sequence of tags. It then identifies the index of the best final state (POS tag) by finding the state with the highest probability in the last position of the trellis matrix using np.argmax. This state index is added to best_path. Next, it iterates in reverse through the positions (words) in the sentence, starting from the second-to-last word to the first word. For each position, it retrieves the index of the previous state (POS tag) from the pointers matrix, which indicates the state that led to the maximum probability for the current state at that position. These state indices are successively added to best_path in reverse order. The resulting best_path list contains the sequence of state indices representing the most likely sequence of POS tags for the input sentence. Finally, it returns these indices to their corresponding POS tags using the S list. It returns the predicted sequence of POS tags as the output of the Viterbi decoding algorithm.

Then I calculated accuracy by using accuracy = correctly_predicted_labels/total_labels

What is the accuracy of the dev data?
Accuracy on dev data after replacing the unknown words in the sentences with < unk > tags before prediction and using a smoothing of 1e-6 for all the probabilities is **0.94754.**

I created a function 'viterbi_on_test_data' to create predictions for test.json and stored the predicted labels and the sentence and index information in the file **'viterbi.json.'**