CSCI 544 – Applied Natural Language Processing

Homework 2

Rishi Kiran Reddy Nareddy
USC ID: 1582852239

Prerequisites:

- Python version: 3.9.12
- Packages used:
  - import json
- Other instructions on how to run the code are furnished in the README file.

Task 1 -> Vocabulary Creation

The vocabulary has to be created from the training data provided. The training data is read and stored into a list. This list stores the training data in line-by-line fashion. For the words alone in the data a separate list is created and a corresponding dictionary is created to store the occurrences of each word. For the threshold, I chose 3 to be the value which means that every word must have repeated at least 3 times. This is done by the following code:

```
threshold_freq = dict((k, v) for k, v in freq_for_vocab.items() if v >= 3)
```

All the remaining words which occur less than the threshold is given an unknown (‘<unk>’) tag. Then the dictionary is sorted in descending order based on the occurrences which includes the frequency of the unknown tags as well.
Entire sorted dictionary is written into a file, in the given format, named ‘vocab.txt’.

For the threshold value 3,
The vocabulary has 16919 words without the <unk> words.
The <unk> occurred for about 32537 times.

Task 2 -> Model Learning

I calculated the transition and emission probabilities by iterating over the lines in the training data. Whenever there is ‘.’, I took the next state as ‘START’ which denotes the starting of a new sentence.

I created two new dictionaries:

emission{}: Which stores the word and tag combination as a key, and number of times the combination occurred in the training data.
transition{}: Which stores the next tag and current tag combination as the key, and value is the number of times the combination occurred in the training data.

The other dictionary tag_freq{} stores the number of times the particular tag has occurred in the training data.

The two other dictionaries are:

emission_probability{}: Stores the emission probabilities.

Transition_probability{}: Stores the transition probabilities.

The emission and transition probabilities are calculated by using the follow formulae:

transition_probability(s_next, s) = transition(s_next, s) / tag_freq(s)

emission_probability(x, s) = emission(x, s) / tag_freq(s)

The dictionaries transition_probability{} and emission_probability{} are together written into a file 'hmm.json'.

Total number entries in Transition: 1394
Total number entries in Emission: 50145

Task 3 -> Greedy Decoding with HMM

Every time I checked if the end of the sentence has reached. If yes, I assign the 'START to prev_tag. If no, prev_tag takes the value of the previous tag in the sequence.
The dictionary tag_freq{} contains the frequency of each tag, hence the keys of the dictionary are the unique tags.

We iterate through the unique tags and check if the tag and the prev_tag has a transition probability and store in a variable. For the tag, we check if the emission probability has value for the current word and the tag and store it in a variable.

We multiply the two probabilities and check it with the max_prob variable which is set to negative infinity. If the product of probabilities (prob) is greater than the max_prob, we assign prob to max_prob and the tag to be the best_tag. We assign the best_tag to the result.

Pseudo code:

```
If len is of a line in data > 0:
      Max_prob = negative infinity
      Prev_tag = last tag of the result list
      Best_tag = ''
Else:
      Prev_tag = 'START'
For curr_tag in tag_freq:
      If (curr_tag,prev_tag) in transition_probability:
            t is associated probability
      if (word,tag) in emission_probability:
            e is associated probability
      prob = e*t
      if prob > max_prob:
            max_prob = prob
            best_tag = tag
```

```
append the word and associated best_tag to the result
return the result
```

When we get the predicted tags for each word in the sentence for the development data, we compare it with the actual tags to get the accuracy.

Accuracy = total of correct predictions / length of the result tag list.

For this greedy decoding algorithm, the accuracy I achieved is `0.9344909234411997` for the development data. We perform the same greedy decoding for the test data and write the result into the file named 'greedy.out' with the given format.

Task 4 -> Viterbi Decoding with HMM

I first populated the entire DP table with dictionaries. Each dictionary represents the product of the emission and transition probabilities of each and every word in the sentence of the development data. If the emission or transition probability for a combination is not present in either of the dictionaries, I took the default value as 0.000000001. For the first word of the sentence we take the previous tag as 'START'. For every other word we create a dictionary within the DP and populate.

Pseudo code:

```
if word first word of a sentence:
        for tag in tag_freq{}:
                if (tag,'START') in transition_prob:
                        t = transition_probability
                        if (word,tag) in emission_prob:
                                e = emission_prob[(word,tag)]
                                dp[index][tag] = e*t
                        else:
                                dp[index][tag] = 0.000000001
                else:
                        dp[index][tag] = 0.000000001
        index+=1
if word is not the first word in the sentence:
        add a new dictionary to the dp
        same steps as above.
        dp[index][tag] = the maximum value of either the previous row tag
                          multiplied by the emission and transition
                          probabilities or the present index and the tag.
```

After we populate the DP, we have to take the max probability of the last word (.) and find Which entry in the previous row gave the max value to the last word. Hence, we backtrack to find the sequence of tags having the highest probability. We reverse the tag sequence since we started from the ending of the sentence to match with the actual sentence.
We populate the backtrack the same we did for DP and do the following,

Psuedo code:

```
if previous row has max value i.e., index-1 than current row i.e., index:
        dp[index][tag] = dp[index-1][key]*e*t
        backtrack[index][tag] = key
tag_seq is a list to have the tag sequence achieved by backtracking
final_tag stores the max value in the dictionary dp[-1] and is added to the
tag_seq list
iterating through the last index of the dp until it reached 0:
        if final_tag in backtrack:
                final_tag = backtrack[i][final_tag]
                add final_tag to the tag_seq
        else:
                tag_seq.append('.')

Reverse the tag_seq to obtain the correct order
We add the tag_seq of each sentence to the final_dp and return final_dp.
```

For the Viterbi Decoding algorithm, I achieved an accuracy of `0.9475517576346305` on the development data. We perform the same Viterbi Decoding algorithm on the test data and write the result into the file named ‘viterbi.out’ with the given format.