TASK 1

Modify data

I created a script that can modify the dev, train, and test files so that they are replaced with pseudo words and <unk> if necessary. There exists a function “wordToPseudo” that checks if the words in a particular file match a format of the designated pseudowords. If there is a format that matches the word will be replaced with the correct pseudo word. There is also a segment that calculates the frequency of each word and if a word is rare it is replaced with <unk>. The modified files are then saved so they can be used later.

Create Vocab

This script is pretty straightforward, it takes in the modified train data as the input and creates a vocabulary based on that data. As per the instructions, the <unk> word is at the beginning of the vocabulary and is followed by the rest of the words in descending order. It is important to note the use of pseudo words that also are in the vocabulary but don’t directly come after <unk>.

What threshold value did you choose for identifying unknown words for replacement?

- **Threshold of 2**

What is the overall size of your vocabulary, and how many times does the special toke “<unk>” occur following the replacement process?

- **22,016**

TASK 2

For learning the hmm I first state the pseudowords that I’m considering and instantiate sets for both the states and observations for easier lookup. The pseudowords are as follows "twoDigitNum", “fourDigitNum", "initCap", "containsDigitAndSlash", "containsDigitAndPeriod", "abbreviation", and "<unk>".

Then I populate the sets with the correct information and create indices so that it is easier to access the positions for states and observations in the future. Next is creating the matrices for the initial state, transition, and emission counts and populating them.

Then I calculate the initial state, transition, and emission probabilities while also using a smoothing technique for increased accuracy. These calculations will be held in their respective dictionaries. Lastly, I populate the model with these dictionaries and write them to the designated output file.

How many transition and emission parameters in your HMM?

- **Transition: 2,025**
- **Emission: 990,765**

TASK 3

Similar to the last task I keep track of the state indices, create the matrices for transition and emission and then populate those matrices with probabilities. The probabilities of the initial states are stored in a numpy array.

For the greedy decoding algorithm I have to handle the unknown words. These are essentially words found in the file that don't appear in the vocabulary. These words are replaced with "<unk>". It is important to compute the first tag for this algorithm because it will be used to calculate the following tags. This process includes calculating the product of the transition probabilities from the previous tag and the emission probabilities for the current word. After this is done it finds the index of the max value in the product and appends this max value to the tags list. (Note: this max value at a given index corresponds to a word) Once all tags are computed that will be held in a list that will be returned.

I use the development data to evaluate the accuracy of the model in order to experiment and make the necessary adjustments. Last, I make the predictions on the test data and output a file with the predicted tags and the original sentences.

* I used numpy and matrix operations for greedy decoding because other approaches had a much longer runtime.

What is the accuracy on the dev data?

- **90.74%**

TASK 4

The viterbi approach was very similar except I didn't use numpy or matrix operations. This is because when I implemented it there wasn't an improvement in accuracy and the runtime actually increased. I also took the approach of comparing the files data to that of the vocabulary by using the emission data. There is a function "checkEmission" that checks each word in the sentence and checks if it appears in the emission probabilities. If it doesn't that means it is not in the vocabulary and the word is replaced with <unk>.
The main difference of this script is obviously the decoding method, here we use the viterbi algorithm.

There is a list that holds the probabilities of the most likely path and a list that holds the states that have been selected. We then have to initialize the first observation. This is done by calculating the initial path probabilities for each state using the initial probabilities and emission probabilities of the first observation and taking the max instance.

There is then the recursive portion of the algorithm where the most likely path probabilities and the respective states are calculated. Then for each state and observation the max log probability is computed from every possible previous states. The max probability is stored in the first list and the state that reached the max is stored in the second list.

In order for the algorithm to stop the max probability of the last observation must be found.

Lastly, in order to construct the path we start from the last found state. We then iterate and find the states that lead to the next state in the most likely path (use the second list). Once this is found it is inserted at the beginning of the path list and once all iterations are complete the path list is returned.

We then calculate the accuracy of the model on the dev data and predict the tags of the test data and output these predictions to the designated file.

What is the accuracy on the dev data?

- **92.23%**