**BBM497: Introduction to Natural Language Processing Lab.** **(Due: 20/04/20)**

# Submission Assignment #2

*Instructor:* Burcu Can *Name:* Utku İpek, *Netid:* 21627356

**Abstract**

In this assignment, the goal is to build a Hidden Markov Model for the named-entity recognition task and also to decode the sequence of observations by determining the corresponding hidden state sequences implementing the Viterbi algorithm. Below two sections show that the data structures I have used and the way of my implementation to make the program efficient in terms of time and space for each task, and the final section shows the calculation of the accuracy which is used as an evaluation metric for this assignment.

## 1 Task I: Building A Bigram Hidden Markov Model

To build a Hidden Markov Model, first, I have implemented the *dataset()* function to read the training data which returns a *list* type that hold the sentences. While reading the training data, I have taken words and tags as pairs to avoid using any extra space. I've also added the beginning and end of the sentence tokens for each sentence. Then, I have implemented the *HMM()* function which takes the list of the sentences as a parameter. In this function, I've declared three *dictionary* types: one to hold unigram tag counts, one to hold bigram tag counts, and one to hold word-tag pair counts. The first dictionary is used in the calculation process of both transition and emission probabilities. The second dictionary is used to calculate transition probabilities, and the third dictionary is used to calculate the emission probabilities.

In the *HMM()* function, to calculate the transition probabilities, I've implemented a helper function, and also instead of using smoothing, I have implemented the *deleted interpolation* algorithm. I've calculated two weight values, one for unigram and one for bigram, then I've created the transition probability matrix. An important note here is that for the initial probabilities, I've used the first row of that transition matrix which contains the probabilities of each tag given beginning token, then I've deleted that row from transition probabilities. The data structure I've used for the transition probabilities is a *dataframe* of Pandas library. The reason is that it is easy to visualize and see the relation between items with a dataframe.

Again, in the *HMM()* function, to calculate the emission probabilities, I've implemented a helper function. This time, I have used *Laplace smoothing.* The data structure that holds the emission probabilities is a *nested dictionary.* First keys of the emission dictionary are the tags, and the second keys are the words.

The last point of this task I want to mention is that the data structures I've declared to hold transitions, emissions, tag counts etc. are assigned to their class variables. This helps me developing and debugging the program by seeing them at any time.

## 2 Task II: Implementing Viterbi Algorithm

First of all, the most challenging part for me was this task of the assignment. This algorithm is an instance of dynamic programming and helps decode the below formula.

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n|w_1^n) \approx \underset{t_1^n}{\operatorname{argmax}} \prod_{i=1}^{n} \overbrace{P(w_i|t_i)}^{\text{emission}} \overbrace{P(t_i|t_{i-1})}^{\text{transition}}$$

The *viterbi()* function I've implemented takes four parameters: a single sentence from test data, initial probabilities, transition probabilities and emission probabilities. In the function, firstly, I have declared a state list which contain states, then I've created the *Viterbi* and *backpointers* matrices as *numpy* arrays. Then, I've created the initialization part of the algorithm using first observation of the test sentence, initial probabilities and emission probabilities. After that, from second observation to the last one, for each state, I've filled the *Viterbi* and *backpointers* matrices with appropriate values using the previous column of the *Viterbi*, transition probabilities and emission probabilities. There are three important points, the first one is that if a key error occurs in emission dictionary, to handle it, I have calculated the smoothed emission probability of the word that causes the error using *try-except* block. The second one is that, I've passed the initial and transition probabilities to the function not as *pandas dataframes* but as *numpy arrays* for the time efficiency. The last one is that to avoid underflow and again to use the time efficiently, I have used the log probabilities in this function.

After that the both *Viterbi* and *backpointers* matrices are filled, I've implemented the backtracking part using those matrices and I've created a list of path of tags indices. Then, finally, I've converted this list to the path list that holds tag as strings. The *viterbi()* function returns the most probable tag sequences for the given sequence of observations.

## 3 Task III: Evaluation

The last task of the assignment is to evaluate the model I've created. To do that, first, I've read the test data using the same *dataset* function, but this time it separates the words and tags and returns two lists: list of test sentences and list of gold sequences. After I've read the test data, I've implemented a function which takes every test sentence and sends it to the *viterbi()* function. This test function returns the list of the tag predictions of each sentence. Then, to calculate the accuracy, I've implemented the *accuracy()* function which compares each tag sequence with each gold sequence and returns the accuracy value. This function uses the below formula to calculate the accuracy.

$$A(W) = \frac{\#of\_correct\_found\_tags}{\#of\_total\_words}$$

## 4 Conclusion

The time the program takes is approximately 8 seconds and the accuracy value I've obtained is approximately 91%. Using *deleted interpolation* algorithm for the transition probabilities and using *Laplace smoothing* for the emission probabilities have helped me getting that accuracy. Of course, the accuracy value can still be improved by using good strategies for handling unknown words, taking the suffixes of the words into the account and building the model not for bigram but for trigram.