# CSCI 544- Applied Natural Language Processing

## Home Work 1


- Sai Kumar Reddy Peddholla
- USC ID: 5978435849


## Prerequisites:

- Python version: 3.11.0
- Mentioned below additional packages used:
  **import sys**
  **import operator**
- I have mentioned the instructions to execute the code in README file.

## Task 1 – Vocabulary Creation:

Reading the input **train** file line by line, created a dictionary to store all the words and its frequency. Created a new dictionary, by filtering all the words with frequency more than 3 and considering every other word as <unk> (unknown). Sorted the dictionary in descending order based the frequency of the words.

Traversing filtered dictionary, created a vocab file to store all the word types and its frequencies along with the index (started with index 1 and incremented it for every new word).

```
Threshold for unknown words: 3
Total size of vocabulary: 13750 (Not including <unk> word type)
Occurrence of Unknown <unk>: 42044
```

## Task 2 – Model Learning:

Traversed the **train** file line by line to calculate the transition and emission occurrences.

Created following dictionaries to store values which will help to compute transition and emission parameters:

- Transition count: Dictionary to store the number of occurrences of transition from start s to state s'. Created a key of both states using **~** (tilde) separator; corresponding key to store occurrences of s -> s' is "**s ~ s'**".
- Emission count: Dictionary to store the number of occurrences of emission from state s to word x. Created a key of both states using **~** (tilde) separator; corresponding key to store occurrences of s -> x is "**s ~ x**".
- Tag Count: Dictionary to store the number of occurrences of each state (tag).

Used below formulas to calculate transition and emission probabilities:

Emission ( s ~ x ) = Emission Count [ s ~ x ] / Tag Count [ s ]

Transition ( s ~ s' ) = Transition Count [ s ~ s' ] / Tag Count [ s ]

Used **<blank>** tag as new line starter tag; Considered **<blank>** tag to compute the tag of first word. The occurrences of **<blank>** tag would be the number of sentences in the train data.

Combined both emission and transition dictionaries to a single dictionary and dumped the data into hmm.json file.

Note: Replace unknown words with **<unk>** tag when found.

**JSON Structure**:

{

transition: {

"(s , s')" : <float value>;

"(s , s')" : <float value>;

"(s , s')" : <float value>;…….. so on (1392 key value pairs)

};

emission: {

"(s , x)" : <float value>;

"(s , x)" : <float value>;

"(s , x)" : <float value>;…… so on (19626 key value pairs)

}

}

```
Number of transition parameters: 1392
Number of emission parameters: 19626
```

## Task 3 – Greedy Decoding with HMM:

Used dev data to calculate the accuracy of the model. Used two variables to store the number of correct matches and number of mis-matches predicted by the hmm model using greedy approach.

Calculated accuracy using below formula:

Accuracy = ( correct match )/( correct match + mismatch )

Used a variable to store the previous state (tag). Initially the previous state would be **<blank>** for each sentence. Traversing each word in a sentence to predict the tag of that corresponding word; we would

only consider previous state and current word; check for all possible current state to maximize the probability.

```
Predicted tag = null

Max probability (mp) = 0

For (cur state) in (possible states):

      Transition probability (tp) = 0.0000000000001 if ( prev state -> cur state ) is not present in transition otherwise initialize with transition["prev state ~ cur state"].

      Emission probability (ep) = 0.0000000000001 if ( cur state -> cur word ) is not present in emission otherwise initialize with emission["cur state ~ cur word"].

      Check if cur prob ( tp * ep ) > mp then:

            mp = cur prob; predicted tag = cur tag
```

In this way we predict the tag of each word in a sentence.

Note: We have given small default values `0.0000000000001` for tp and ep because the product would become 0 and we wont be considering any tag.

Similarly performed the same operation to predict the tags of words in the sentences in test data. Output is similar to train file format. The file would be created with the name **greedy.out** in the destination folder path provided as an argument while running the code.

```
Greedy Algorithm accuracy: 0.9196542407868374
```

Note: Replaced unknown words with **<unk>** tag when found.

## Task 4 – Viterbi Decoding:

Used dev data to calculate the accuracy of the model. Used two variables to store the number of correct matches and number of mis-matches predicted by the hmm model using dynamic programming approach.

Calculated accuracy using below formula:

Accuracy = ( correct match )/( correct match + mismatch )

Used a variable to store the previous state (tag). Initially the previous state would be **<blank>** for each sentence. Traversing word by word in a sentence to create a table of size (number of words) * (number of states). Using previous state as **<blank>** filled the first column with probabilities of all states.

```
prev state = <blank>

For (cur state) in (possible states):

      Transition probability (tp) = 0.00000001 if ( prev state -> cur state ) is not present in transition otherwise initialize with transition["prev state ~ cur state"].

      Emission probability (ep) = 0.00000001 if ( cur state -> cur word ) is not present in emission otherwise initialize with emission["cur state ~ cur word"].
```

```
Dp[cur state][cur word] = cur probability (tp*ep)
```

After filling the first column, we will keep on filling the complete table with probabilities with all possible pairs of current word and state with all possible previous states.

```
For (cur word) in (sentence):

For (cur state) in (possible states):

    Emission probability (ep) = 0.0000000000001 if ( cur state -> cur word ) is not present in emission otherwise initialize with emission["cur state ~ cur word"].

    Max probability (mp) = 0

    For (prev state) in (possible states):

        Transition probability (tp) = 0.0000000000001 if ( prev state -> cur state ) is not present in transition otherwise initialize with transition["prev state ~ cur state"].

        Check if cur prob ( tp * ep ) > mp then:

            mp = cur prob

    Dp[cur state][cur word] = mp (max probability)
```

After filling up the entire table we will back track the table from last column to first column to computer the sequence of tags which gave max probability in the last column.

First, we will compute max probability in the last column

```
    Max probability (mp) = 0

    Next state = -1

    For (cur state) in (possible states):

        If mp <= Dp[cur state][last word]

            mp = Dp[cur state][last word]

            next state = cur state
```

Note: If the last column maximum probability is too small, we are initializing last tag as '.' (full stop).

After predicting the last state, we will traverse table from last but one word to first word to compute sequence:

```
    For (cur word) in (all words): [Traversing reverse]

        Prob diff = 1

        Pred state = null

        For (cur state) in (possible states):

            Transition probability (tp) = 0.0000000000001 if ( prev state -> Next state ) is not present in transition otherwise initialize with transition["prev state ~ cur state"].
```

```
                    Emission probability (ep) = 0.0000000000001 if ( next state ->
next word ) is not present in emission otherwise initialize with emission["cur state ~
next word"].

                    If Prob diff > Dp[cur word][cur state]*(tp*ep) – next:

                          Next state = cur state

                          Prob Diff = Dp[cur word][cur state]*(tp*ep) – next

              Pred state = Next state
```

Here we try to compute the corresponding column from which we have generated max probability in the next column. Since the values of probability are small, there is a chance of overflow and approximation. So I have used 'Prob diff' to consider the nearest column which could compute the required probability by taking difference (Current column value * transition probability * emission probability – Next Column value).

Note: We have given small default values `0.0000000000001` for tp and ep because the product would become 0 and we wont be considering any tag.

Similarly performed the same operation to predict the tags of words in the sentences in test data. Output is similar to train file format. The file would be created with the name **viterbi.out** in the destination folder path provided as an argument while running the code.

`Viterbi Algorithm accuracy: 0.9364710704283079`

Note: Replaced unknown words with **<unk>** tag when found.