CSCI544: APPLIED NATURAL LANGUAGE PROCESSING

HOMEWORK ASSIGNMENT – 2 – 02/08/2023

Part-Of-Speech Tagging

Name: Reuel Samuel Sam
Email: rsam@usc.edu

I. Problem Statement

This assignment has tasked the students with computing the Part-of-Speech tags for a given corpus by calculating the transition and emission probabilities before decoding using Greedy and Viterbi decoding algorithms.

II. Libraries Used

For the assignment, Python 3.10.8 and the following libraries were used:

- pandas
- numpy
- sys
- json

III. Task 0: Reading the Data

For reading the data from the given files, a function has been written with a few parameters. Based on the parameters, the data returned will differ. The parameters are:

- filename : takes the name of the file that is to be read
- get_vocab : boolean variable that indicates whether the list of words should be created for the given file
- separate_sentences : boolean variable that indicates whether the data should be a single dataframe containing all sentences, or if it should be a list of dataframes where each dataframe consists of only one word
- replace_unknown : Boolean variable that indicates whether the words that occur below a certain threshold need to be replaced with "<UNK>" or not
- vocabulary : Dataframe containing words that occur in the vocabulary (formed after removing those words that occur less than a set threshold number of times)
- tags_present : Boolean variable that indicates whether the list of tags need to be returned or not (needed to create .out files)

When data is read, at the start of every sentence, a new entry is made: (0, <START>, <START_TAG>). This is to allow for the initial transition to take place when it comes to the model creation. Additionally, those words that occur fewer than a set number of times (here, 2) in the corpus are replaced with <UNK> tag. Exceptions exist based on the parameters that are passed to the function.

IV. Task 1: Vocabulary Creation

As instructed, a vocab.txt is written after the 'unknown' words are removed. The following have been reported:

- Threshold : 2
- Total Size of Vocabulary after removing Unknowns : 23183
- Count of Unknown tags : 20011

For vocabulary creation, yet another function has been written that takes the following parameters:

- data : dataframe of the corpus that has been read
- vocab_data : Dataframe containing the just the words that occur in the corpus
- count_threshold : Value of threshold where if a word occurs fewer times than the threshold, they are not considered as part of the vocabulary. Set to 2 by default

V. Task 2: Model Learning - Calculating Transition and Emission Probabilities

- Number of transition parameters (Non-Zero) : 1416
- Number of emission parameters (Non-Zero) : 30305
- Number of transition parameters (Zero included) : 2116
- Number of emission parameters (Zero included) : 1066464

For this task, a class has been written containing a few methods for the sake of modularity and reusability. The class constructor takes in optional parameters (done in case data is being loaded in, and no training is required). Before training is started, helper data structures are set up. First, for fast calculations of counts, the data is converted into a dictionary since it has constant look up times. Additionally, the count of each tag, the list of tags and list of words are all maintained in separate structure. "<START>" is added to the word list, and by extension, to the vocabulary (therefore, size of vocabulary becomes 23184. As a result, the number on non zero emission parameters: 23184*46 = 1066464).

In addition to these, two dictionaries – one each for transition and emission probabilities – are also initialized to maintain the counts of each pair of <tag, tag> or <tag, word>. **For any**

combination that does not appear in the corpus, their probability is set to 0. Contrary to the implementation mentioned in the instructions, this dictionary is maintained as a dictionary of dictionaries with the following hierarchy:

Transition Probability	Emission Probability
tag_1: { tag_1: val_{11}, tag_2: val_{21} …}, tag_2: { tag_1: val_{12}, tag_2: val_{12}… } …	tag_1: { $word_1$: val_{11}, $word_2$: val_{21} …}, tag_2: { $word_1$: val_{12}, $word_2$: val_{12}… } …

Here, transition_prob[tag1][tag2] signifies the probability that tag2 occurs immediately after tag1. Similarly, emission_prob[tag1][word1] signifies the probability that word1 occurs immediately after tag1. This representation is used throughout Tasks 2, 3 and 4. However, before writing to the json file, these dictionaries are converted to the instructed format where all non zero values have been omitted to avoid any forms of confusion.

The other methods defined under this class were used for writing and reading data as well as checking if the sum of probabilities for each tag sum up to 1.

VI. Task 3: Greedy Decoding with HMM

- Greedy Decoding Accuracy on dev_data **:** 93.4870378240544

For this task, a class has been written for the sake of modularity and reusability. The input data for this task is a list of data frames were each dataframe represents a sentence. When it comes to predicting the tags, each sentence is passed as an argument to the predict_sentence function.

In this, for each word, the probability is calculated as the as the maximum product pair of transition probability for the current tag that is being considered given previous tag (initialized to <START_TAG>) multiplied with the emission probability for the word given the tag that is being considered. Once the maximum probability for the word is found, the current tag is set as the previous tag and the process is repeated for the each word in the sentence. At the end of computation, lists containing each of the tags for each of the words are returned.

For accuracy score calculation, an iterator goes through the entire list of tags for the whole corpus and a comparison is made with the actual tags. The accuracy is then calculated as the number of matches divided by the total number of words that the tags have been predicted for.

For the test data, the tags are calculated and then the predictions are generated into an output file labeled greedy.out in the required format.

VII. Task 4: Viterbi Decoding with HMM

- Viterbi Decoding Accuracy on dev_data : 94.76883613623946

For this task, a class has been written for the sake of modularity and reusability. The input data for this task is a list of data frames were each dataframe represents a sentence. When it comes to predicting the tags, each sentence is passed as an argument to the predict_sentence function.

Since this is a dynamic programming approach, a memoization matrix has been initialized with zeroes and labeled as OPT. In addition to this, a new matrix has been initialized with zeroes for backtracking purposes. Both matrices have the same shape (no. of tags, length of sentence). Additionally, for the first word in the sentence, the values of OPT matrix are initialized as the product of transition probability for the tag being considered, given <START_TAG> tag and the emission probability for the word, given the tag being considered. Since there is no previous word, the backtrack matrix is left untouched as it has already been initialized to zero.

Now, for every other word in the sentence, we calculate the corresponding OPT value for the word and the candidate tag as the maximum product that is found by multiplying three terms:

- OPT value for previous candidate tag and previous word
- Transition probability of current candidate tag, given the previous candidate tag
- Emission probability of the current word, given the current candidate tag

Once the pair of tags that give the maximum probability are found, the OPT value is set to the probability value while the backtrack matrix is updated such that the entry for (current word, current tag with max probability) is the index for the tag for the previous word that gave this maximum probability.

Once the OPT matrix is filled up, the algorithm backtracks through the backtrack matrix starting from the last word's corresponding max tag. To find the tag for the previous word, the value in the backtrack matrix for the current word will point to the index for max tag for the previous word. This is then repeated for all the words in the sentence till the first word. At the end, a list is returned that contains tag for the sentence that is being considered.

For accuracy score calculation, an iterator goes through the entire list of tags for the whole corpus and a comparison is made with the actual tags. The accuracy is then calculated as the number of matches divided by the total number of words that the tags have been predicted for.

For the test data, the tags are calculated and then the predictions are generated into an output file labeled viterbi.out in the required format.