CS6320 Assignment 2

https://github.com/cs6320-501-fall22/assignment2-group15

Group15 Akshaiy Ramlinkam axj210002 Madhumita Ramesh mxr210041

1 Introduction

The assignment works on building a POS tagger to generate an entire sequence of predicted tags for real-world data. The approach is motivated by HMM which has the words of the sentence as the **Observable** states and their POS tags as the **Hidden** states. It includes the following components: a **set of possible tags, transition matrix, sequence of words (corpus), and emission matrix**, which will be described in further detail later.

It uses the **Viterbi** algorithm as the decoding algorithm for the testing of the accuracy of the built POS tagger using the training dataset. We have built a transition matrix with **Add-0.01** smoothing by adding 0.01 to the frequency of the number of occurrences of [previous tag, current tag] divided by the (sum of the frequencies for the current tag across all previous tags + the total number of tags i.e. 46). Similarly, the emission matrix is generated using the training corpus by **Add-0.01 smoothing** by 0.01 to the frequency of the number of occurrences of [word, tag] and 199.24 (0.01*19924 columns) to the sum of the frequencies of the tag across all words (denominator).

The accuracy is measured by comparing the number of sequences of tags that match with the predicted sequence and the count of the mismatches at every position is used to calculate the discrepancy in the model. The **less the discrepancy, the better the model** is. The best model derived from testing the validation data (with the **Add-k parameters and threshold for refined vocabulary**) is used to compute the F1 score for the testing data.

2 Data

For this assignment, we have been given two training csv files, one containing a map of words of paragraphs in a key-value pair format (x_train) and the other file (y_train) containing the corresponding POS tags for each word.

There is a similar dataset format (2 files, x_dev, and y_dev) for validation data as well, which will be used for testing, to generate the best POS tagger for the validation corpus.

We have created two list variables, **docs, and tags**, which are used to store the pre-processed data from the given input. docs is a list of lists, each element of which contains a paragraph. Each element in this list is also a list, which contains all the words of the given paragraph, separated by commas. Similarly, tags list is going to store all tags of each paragraph as a list element in tags. So for any word in docs[i][j] from the training corpus, the corresponding tag is also stored in tags[i][j]. Similarly, the given validation data is also split into a list of paragraphs but for this assignment, we have considered all the words together. We have also created a list variable **pos** which stores a list of unique tags used in the corpus (**46** in total).

The training corpus is segregated by keeping a **threshold frequency=1** to handle the case of unknown words. All the words in the corpus with a frequency of 1 are filled into UNK_x buckets (where x = noun or plural noun or verb etc.). From the data, there are a total of **17592** unknown words in the training set. Consequently, the vocab set, which contains a list of all unique words, will also have an additional word **UNK**. Hence, from the given data, there are a total of **19924** unique words (including UNK) in the vocab set.

Given below are the statistics for the number and count of the words for paragraphs across training, validation, and testing datasets.

	Paragraphs	Words
Training	**1387**	**696475**
Validation	**462**	**243021**
Testing	**463**	**236582**

3 Implementations

The main idea of implementation was using Hidden Markov models. **Viterbi algorithm** is used to optimize the HMM to tag the words. So, given a sequence of words in a paragraph, we have to find the most probable sequence of tags associated with each word in that paragraph. The intuition behind using this algorithm for finding the best probable tag comes from the probabilistic determination of a tag for the given word, given a previous sequence. It makes use of the **Naive Bayes theorem**:

$$P(t|w) = \frac{P(w|t) \cdot P(t)}{P(w)}$$

Here, P(t—w) is the probability of a tag being associated with the word, given the word. For comparison purposes, we can normalize by ignoring the computation of the denominator P(w), which will leave us with only the probabilities, P(w—t) and P(t).

The first term p(t) is called the **transition probability** and as the name suggests, it calculates the probability of transitioning from one tag state to another. Here, in a tagging task, we are assuming that a tag will **only** be dependent on a previous tag. In order words, we will be calculating a 2d matrix of **bigrams** with their frequencies.

```
for i in range(len(x_train["word"])-1):
  a,b = y_train["tag"][i],y_train["tag"][i+1]
  transition_matrix.at[a,b]+=1
```

Listing 1: Transition Matrix

A matrix of dimensions **46*46** is generated using **Pandas Dataframe** *(later on converted to **numPy** because of quicker indexing)* where the rows indicate the first tag of a bigram and the column index indicates the second tag (or the current tag) of the bigram. A for loop is used to traverse through each consecutive pair and increment the corresponding frequency on the matrix cell (here indicated by a,b).

The probability is calculated by dividing the cell[i][j] by the sum of the frequencies across the row i for every row cell

$$P(t)[i,j] = \frac{count(i,j)}{\sum_{j=1}^{46} count(i,j)}$$

The second term p(w/t) is called the **emission probability**. It defines the probability/ chance of the occurrence of a word in a sentence given the condition that the tag of the word for that position is given. The given matrix will be **much larger in size** in comparison to the transition matrix.

```
for i in range(len(docs)):
  for j in range(len(docs[i])):
      word = docs[i][j]
      tag = tags[i][j]
      if word not in unk_words:
          emmission_matrix.at[word,tag] += 1
```

Listing 2: Emission matrix

The matrix will be of the dimensions **19924*46**, generated using the **Pandas DataFrame** *(later on converted to **numPy** because of quicker indexing)*. Here rows indicate the unique word from the training dataset while columns indicate all the values of tags. A for loop is maintained with i as the loop variable traversing through each word in the docs list and fetching the tag from the corresponding index in the tags list. Using the tag t that we receive from the tags list, we can increment the frequency of the matrix at the (word, t) cell. The probability is calculated by dividing the cell[i][j] by the sum of the frequencies across column j for every column cell.

Logic: The probability of a tag associated with a word is hence calculated by maximizing the product of the emission and the transition probability at each word and the corresponding tag is noted for that word. At every word, we also have to make sure that the whole sequence probability is maximized.

After the matrices are generated using the training corpus, it has to be validated using the validation dataset to identify the best combination of parameters to get high accuracy. The concluded model will hence be decoded by Viterbi. The motivation behind choosing Viterbi comes from the underlying fact that HMM is a more time-consuming algorithm. HMM is dependent on every state and its corresponding observed object.

The purpose of the Viterbi algorithm is to make an inference based on a trained model and some observed data. It works by asking a question: given the trained parameter matrices and data, what is the choice of states such that the joint probability reaches maximum? In other words, what is the most likely choice given the data and the trained model? The following equation represents the highest probability along a single path for first t observations which ends at state i.

$$\omega_i(t) = \max_{s_1,\ldots,s_{T-1}} p(s_1, s_2 \ldots . s_T = i, v_1, v_2 \ldots v_T | \theta)$$

Here, in the assignment, we are maintaining two matrices; score_matrix and tag_tracking_mat. The first matrix, **score_matrix** contains the maximum probability of the multiplication of the transition and the emission probabilities at each step i of the sentence. The following code contains the snippet of the probability multiplication for each cell [i][j].

```
score_matrix[curr_tag_idx][j] = max (score_matrix[old_tag_idx][j-1]
              + math.log(transition_matrix[old_tag_idx][curr_tag_idx])
              + math.log(emission_matrix[train_wor_idx][curr_tag_idx]))
```

Listing 3: Probability Calculation at each word

In the above code snippet, **score_matrix[old_tag_idx][j-1]** represents the previous word probability, given its tag for every word j-1 from 1 to the total number of words in the corpus;

math.log(transition_matrix[old_tag_idx][curr_tag_idx]) indicates the log of the probability from the transition matrix which represents the cell [previous tag][current tag] for every previous tag old_tag_idx from 1 to the total number of tags, and it is calculated across all the current tags even from 1 to the total number of tags. This gives us the maximized probability of all the current tags, which are calculated by the transition from the best combination of the previous tag and;

math.log(emission_matrix[train_word_idx][curr_tag_idx]) shows the log of the probability of the current word, derived by its index in the training word corpus train_word_idx; given its current tag curr_tag_idx, and the index of the word is taken from the testing corpus.

The **tag_tracking_mat** matrix, on the other hand, contains the tags which will help to backtrack the mode likely hidden path. It is filled according to the values populated in the score_matrix matrix as explained previously. For every cell [i][j] in the score_matrix, we calculate the maximum sum of the three probabilities and assign that value to the cell[i][j]. The corresponding value of the previous tag, for which the value of score_matrix was highest for [i][j] is stored in tag_tracking_mat[i][j]. This helps us keep track of the previous tag as we move on to the next layer.

4 Experimental Design and Evaluations

Through trials, we have observed that it takes around **9-10 mins** to obtain the results using the **Viterbi** algorithm.

We have also incorporated a methodology to handle unknown words. We have tried incorporating a few methods and tried validating before we stuck with Viterbi as the best suitable method for the validation data.

One of the methods was using an **Open Vocabulary** concept. Here, we replace all the words with frequency=1 in the training data with UNK . Now, whenever we encounter an unknown word in the validation and the testing corpus, we take the maximum probability from the transition and emission matrices for the unknown words with tags and assign the appropriate tag for the same.

Another method involved using **suffixes** as a medium to differentiate between the different unknown words. We maintained a list of common suffixes/patterns to group words into - **nouns, plural nouns, verbs, adjectives, adverbs, capitalized words, capitalized plural words, and words with uncommon character sequences** and checked for each unknown word to assign the corresponding tag and emission probability for the calculations. We combined both methods to increase our unknown word accuracy and implemented it in the Forward pass of the Viterbi algorithm section. Here, the index of the word (UNK if below the threshold, otherwise the actual word) in the refined vocab set is used to display the value from the emission matrix in the score_matrix formula.

5 Results

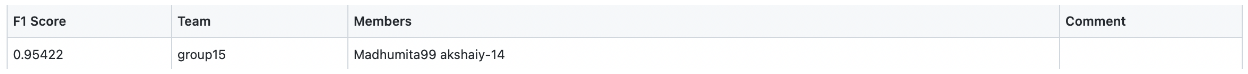
The POS tagger model is built using the **Open Vocabulary** concept and inclusion of **Suffixes/Patterns** of Unknown word handling with the **Viterbi** algorithm and the results on the validation and testing data are as follows:

1. Accuracy on the validation dataset: 231253 out of 243021 values matched (**95.1576%**)
2. F1 score on the validation dataset: **0.9521**
3. F1 Score on the Test data: **0.95422** (from the leaderboard)
4. Accuracy on Unknown words: 6298 out of 8569 values matched (**73.8899%**)

The **variations that we had experimented** with on both the training/validation and test data include:

1. Splitting the training dataset into a **list of paragraphs** instead of treating the whole corpus as one. This way, the validation accuracy was lesser.
2. Taking a **greedy** approach by only maximizing the multiplication of the probabilities at each word instead of taking the whole sentence into account. Here, we didn't handle UNK words - just considered the transition probability when a new word was encountered. This resulted in **88%** accuracy in the validation set.
3. Using a **small epsilon value** (1/total number of words in the corpus) as the emission probability instead of zero when an unknown word is encountered. This method works as a form of Laplace smoothing.
4. We tried **different values for k** in the Add-k smoothing of transition and emission matrices. As mentioned above, as k decreased from 1 to lower decimal values, the results got better until it reached a saturation point at around x¿10e-3. The main reason for this was because the probability of relevant tags didn't change much if k was 1 or 0.01 on smoothing, but for the tags with a score of 0, the scores were 10 times lower when k was 0.01 when compared to k=1. So this makes a significant difference while computing the Viterbi matrices in the end.
5. Creating a new **refined vocab set with one "UNK"** word replacing all the words with frequency=1. The performance was still good with **92%** accuracy in validation data.
6. Creating a new **refined vocab set with "UNK-x"** words - same as the variation but with the inclusion of suffixes/patterns into the "UNK" words.

Leaderboard Score F1 score: **0.95422 for Group 15** - Madhumita99 and akshaiy-14



F1 Score	Team	Members	Comment
0.95422	group15	Madhumita99 akshaiy-14	

6 Analysis

The model is tested with a variety of different parameters as mentioned below and each of the given factors has a contribution to showcasing error in predicting the tag sequence:

1. **Different values of k** for Smoothing of Transition and Emission matrices - It was observed that lower values of k for both matrices produced better accuracy results and it gets saturated below a certain limit (for eg. below 0.0001). The improvement of the model is hence, only up to a certain value and it varies for different types of datasets.
2. **Unknown word trend analysis**: The common suffixes which occur for words of different tags are noted and used to compare in order to assign the correct UNK word during the validation data. There is an error pattern observed here as it isn't likely to be able to include all the edge cases while predicting a tag using this technique. It can be further improved by considering niche cases that aren't really common but contribute to the errors.
3. **Bigram HMM**: The probability of a tag depends only on the previous tag(bigram HMM) that occurred rather than the entire previous tag sequence i.e shows Markov Property. It isn't as effective since there can be examples where the current tag is based on a tag of a word that might be directly previous to it. It can be improved by using n-gram HMM and observing the best value of n for the given training dataset.
4. **Threshold for UNK**: Setting the ratio of words to UNK in training data is a key factor as UNK can only be predicted better in the test if it is properly analyzed with ample data from the training set, which can be achieved by increasing the threshold for frequency.

7 Conclusion

From the above assignment, we have inferred the following conclusions:

1. **Lower values of k** (from the assignment we have inferred that k=0.01 gives better accuracy than k=1) for Add-k Smoothing give a better result in building a POS tagger for the validation and test dataset.
2. Unknown word handling has a variety of approaches to deal with and it depends on the **context and the area of concentration** of the text we are working with.
3. We have made use of the **Open Vocabulary** concept and inclusion of **Suffixes/Patterns** for handling unknown words in the validation data.
4. It takes **much longer times** to compute the score_matrix and tag_tracking_mat in the Viterbi algorithm when they are generated with Pandas Dataframe compared to Numpy. (**NumPy indexing is much faster than Pandas**).
5. **Bigrams** are used to generate the transition and emission matrices.
6. Test data was validated for its accuracy with Viterbi decoding, which was the best model for the validation data, and the **Accuracy for test data was computed to be approximately 95.4%**.

Other

- Both members of the group have contributed equally in conveying ideas and logical concepts and implementing the same using Python. Each has however taken selective sections to take control over to identify the details needed for the implementation of the same. **Akshaiy Ramlinkam** (axj210002) has professional fluency in Python, hence was more comfortable with coding more complex subsections like the Viterbi algorithm and handling unknown words, whereas **Madhumita Ramesh** (mxr210041) has worked on other subsections like building the transition and emission matrices and calculation of accuracy. Both have equally worked on fairing out the documentation as a report.
- The project gave us a practical sense of working with a POS tagger model. It was made challenging with the inclusion of the prediction of data using the built model. It gave us a detailed understanding and clarity as it made us visualize the theory that goes into these concepts. It was an extensive assignment and it took us more than a week to be able to be comfortable with the concepts and python language and implement them with testing. This assignment was more of a difficult task as it involved a lot of repeated computation, which created confusion across different datasets and this subject is relatively new to the group members.