Saishnu Ramesh Kumar

300758706

Project 1 - Statistical Language Modeling

Abstract:

Statistical Language Modeling is a process that develops probabilistic models that can help predict the likelihood of the next word in each sequence based on the context of the previous words. In this project, we are given a sequence of N-1 words, which is an N-gram model to predict the most probable word to follow a sequence. We train the model using text that is obtained from a book of our choice and implement the following functions from the unigram, bigram, and plot distribution of the unique words that have been found within the book.

Introduction:

The book that we will be using in this study is "Alice in Wonderland", written by Lewis Carroll on July 26th, 1951. This book was obtained through the following link: https://www.gutenberg.org/ebooks/11. The basic premise of the story follows the tale of a young girl named Alice who fell down a rabbit hole and ended up in a world of fantasy. From the functions I have implemented, we will be able to dissect important information and results regarding the book itself like the word count, which is around 30,671 and I would also be implementing a function that would show the plot distribution of certain unique words that are found in the text. Some key findings would be the unigram, bigram, and text generation functions where we would obtain important results for this study.

Method Descriptions:

The use of N-gram models can help estimate the probabilities of certain words that are found in the book. The purpose of an n-gram model is to predict the most probable word that will follow along within the sentence. N-gram models are normally trained using text that is provided. They can be used in various Natural Language Processing scenarios from speech recognition to predicting text inputs as seen on our smartphones keyboard.

Several functions for this project have been implemented as per the tasks that were assigned. My approach was to create separate files for each function, and they were all called into one file, the main, to display the necessary outputs requested.

Below are the functions implemented for this project:

1. **readTextFile:** This function takes in a filename as its input and returns the contents of the inputs as a text string. I used fopen, in this case, to open the specific file in read-only mode and the fileID is then stored. The fscan function will then read the contents of the file by using the fileID and indicating that the contents within be read

as a character by using "%c". The text will then be stored in the variable text. The function then ends with fclose where it closes the file, and the "text" variable is returned as an output.

2. **extractWords:** This function takes the text string as the input, and it returns an array of words that have been found in the text. We first must convert the input text to lowercase by using the lower function. Then, we remove all the non-alphanumerical characters from the text and replace them with spaces by using the regular expression function. Lastly, we call the split function to split the results of the text string into an array of words by using the space as the delimiter. The resulting array is then stored in the "words" variables which are returned as an output.

3. **countWords:** This function takes a cell array of strings as the input and returns two outputs, the cell array of unique words, and the vector containing the count of each unique word in the input array. The unique function obtains the array of unique words from the input array of "words" and the resulting array is then stored in "uniqueWords". I created a vector of zeros with the same size as "uniqueWords" by using the zeros function to store in the "wordCount" variable. A for loop is then created to iterate over the unique word in the "uniqueWords" array. For every unique word, the sum function counts the number of times that word appears in the original array of "words" and by using the strcmp function, we can compare each word to the current unique word. We then store the results in the "wordCount" vector. The "uniqueWords" and "wordCount" is returned as the outputs.

4. **countUniqueWords:** This function takes the text string as its input and will return two arrays, unique words found in the text, and a vector that contains the count of each unique word. Once again, the lower and regular expression functions are called to ensure only the text is obtained. I then used the strsplit function to split the text into an array of words and store that array in the "words" variable. By using the cellfun function, it will remove any empty words from the variable and the tilde symbol specifies the isempty function to be the input argument. The unique function obtains the array of unique words from the modified "words" array. The array is then stored in the "uniqueWords" variable. The use of the accumarray function allows us to count the occurrences of each unique word in this case. It takes in the input array of indices that map each word to the unique index in the "uniqueWords" array. The resulting count is then stored in the "wordCounts" vector. The output returned from this are the "uniqueWords" and "wordCounts".

5. **countUniqueWordsWithMinChars:** This function takes in two inputs, the text string and minChars. The function will then return the number of unique words in the test that have a greater length than or equal to the minChars. We first will split the input text using the split function and store it in the "words" variable. The cellfun function applies the length function for each word in the "words" array and extracts the only words that have a length greater than or equal to minChars. It is then stored in "longWords" variable. The unique function obtains an array of the unique words from the "longWords" array and stores it in the "uniqueWords" variable. The length of the uniqueWords is calculated and the "count" variable is output.

6. **unigram:** The unigram function takes in a text string and returns the count of each unique word in the text. The input text is split using the split function and stored in the "words" variable. The unique function obtains the array of unique words from the "words" array and is then stored in the "uniqueWords" variable. Using the zeros function, an array of zeros is created that has the same size as the "uniqueWords" array. It stores the "counts" variable. A for loop is implemented to iterate over each unique word in the "uniqueWords" array. Inside the loop, a sum function counts the number of times the current unique word appears in the "words" array by using strcmp. The count is then stored in the index of the "counts" array which results in the output.

7. **bigram:** For the bigram function, we take in a text string and return the count of each unique adjacent word in the text. Like the unigram, we call the split and unique functions respectively. Then an array of zeros with x-dimensions is created using the zeros function and we store it in the "counts" variable. A for loop is then created to iterate over each word in the "words" array starting from the second word as bigram counts are based on pairs. The indices of the previous and current words are obtained using the find function and the strcmp function matches the words along with their indices in the "uniqueWords" array. The "counts" array is incremented by 1 for the bigram that consists of the previous and current words by using the indices obtained from the find function. The array is then returned to the output.

8. **generateText:** This function will generate a new text for the given number of words that is based on the input text. It takes in three arguments, the input text, starting word, and number of words. First, we will create a dictionary of all the unique words along with the frequency count. Then a transition matrix is created to represent the probability of each word that follows another word. This matrix was made by implementing a for loop that iterates through the input text and counts the number of times each word follows another word. The code will then generate the new text by starting with the given seed word and using that the transition matrix will then predict the next word. The text generated is built one word at a time where the next word is based on the probability of it following the previous word inside the transition matrix. The output returns a string of the text generated.

    a. **Additional constraints:**

        i. Added a constraint in the generateText function to make sure that the generated text contains a specific set of words or phrases. This happens right after the seed word where if the constraint has been specified, the code will check if the starting word fulfills the constraint, if not it will keep selecting a new starting word until it finds the right word. The code will then select the next word that is based on the transition matrix and check to see if it satisfies the constraint, if not the process is repeated until a constraint is found. This will go on depending on the number of words we want to generate.

ii. Added Laplace smoothing after the transition matrix is created to get the probability of a word following another word. This ensures that the words have a non-zero probability and rare words could be generated as well.

9. **plotWordDistribution:** For this function, we are taking the filename as the input. We then call and use the previously mentioned function, file read, lower, regular expression, string split, cellfun, and then find the unique words and their counts. Then we sort the unique words by count in descending order and now we plot the distribution of the 50 unique words by using a bar chart. The x and y axis are labeled with a title given to the graph as well. This output can then be viewed further down this report.

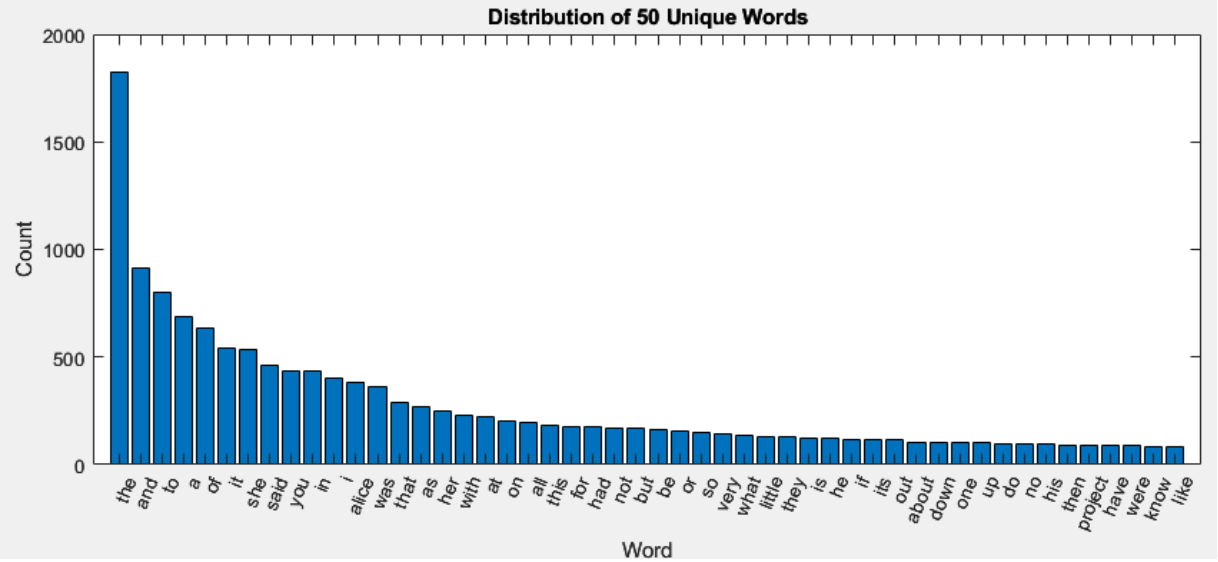
Analysis of Book & Graph Distribution:

Alice in Wonderland is a famous children's book that introduces a young girl named Alice who fell down a rabbit hole one day and ended up in a world filled with fantasy and unique characters. While in Wonderland, she embarks on different adventures with characters she met along the way. The book has turned into a huge success and live-actions movies of it have also been made and spin-offs of the original book as well.

From the code implemented, here is some information regarding the book:

- **Word Count:** 30,671
- **Number of Unique Words:** 3,197
- **Generated Text:** 'alice thought alice s evidence said alice cautiously replied very politely did so she had to receive a mineral i'
- **Most word occurrence: "The"** with a frequency of around 1,800.

The graph below displays the distribution of the first 50 unique words in the text. This was implemented using the **plotWordDistribution** function and is then called in the main file:



As we can observe from the graph above, it has been arranged to go in descending order, which is why the word with the highest count starts from left to right. The word with the highest count is the word **"the"** which reaches around 1,800. Next is followed by the word **"and"** which falls just below 1,000. The graph then keeps decreasing until nearing the end of the graph where it looks like some of the words may have a very closely similar word count, for example, from the word **"out"** to **"one"** and **"up"** to **"no"**. The least occurring word in the graph is **"like"**.

Additional Improvements for Text Generation:

Text generation falls under the Natural Language Processing category whereby the objective of it is to generate text from a given input. They are applied to the real world by helping fill up incomplete sentences or predict the next word we would use. Nonetheless, we cannot exactly depend on the usability of the text generation function implemented in this study as there are many more factors that we would need to implement in order to make it more accurate. To improve text generation, we can consider the following:

- **Meaning of Words:** If we are able to provide the meanings of different words, the text generator could provide better results. It would be able to understand the definitions and when it needs to produce a certain type of word in a certain sentence, it would be able to provide the following. Incorporating the word's meanings will also be able to convey certain types of expressions and emotions. For example, if you were to write a happy text, the generator would be able to come up with relevant words relating to happiness and could suggest them to you for use.

- **Dictionary of Synonyms:** Using a dictionary of synonyms can help the text generator to decide on better words or even detect words that may have similar definitions to each other. It would help to decide on various word usage while still keeping the same meaning and context of what is trying to be conveyed. It may also

help in preventing word repetition to create more engaging sentences and possibly expand someone’s vocabulary.

- **Grammar:** By being able to improve grammar, we can create more detailed and expansive sentences that sound more meaningful. We could create different sentence structures like simple, compound, and complex sentences to even various subject-verb and subject-verb-object agreements. We can also implement grammar rules to ensure that the generator reduces grammatical errors in sentences to make them more readable.

Some of the possible future works to improve my current model would be to possibly add some of these features into what is currently made. This will help improve my model's word generation by allowing it to understand words better and try to use synonyms to prevent word repetition. By also introducing various grammar uses, the model would be able to create much better sentences that would sound more eye-catching to read.