Mini Project 3 Report

Randeep Singh, Michael Pierce
Clarkson University
Potsdam, New York
ransing@clarkson.edu; mipierc@clarkson.edu

1 How to Run

Before attempting to run the provided code, make sure that the following packages are installed for Python: nltk and collections, and re. For the Word Prediction program, make sure to have the following files inside the Word Prediction folder: "big.txt", "Word Prediction.py". Simply run the code from that folder and input a sentence into the command line argument. To exit the program, just type "quit" in an empty line. For the Spelling Correction program, make sure to have the following files inside the Spell Correction folder: "big,txt" and "spellcheck.py". To run the spell corrector simply run the program from the folder. From there just type the sentence you wish to correct and the program will print out the corrected sentence. To quit the program simply do the same as you do from the word predictor program.

2 Problem Statement

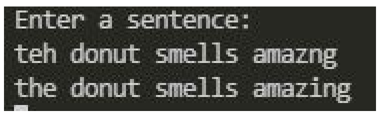
Spell correction and word prediction play a large role in the way we type to friends, family, and colleagues. Our task was to create a creative way to attempt to correct spelling issues and predict the next word a user may type. We divided our this task into separate programs to lessen the complexity of the project. In order to perform tasks such as text prediction and spell correction, we needed a large corpus of data to use as the foundation of each of the programs. As such, in order to gather the a large database of words and common pairs of words, we needed to tokenize our corpus source, in our case known as 'big.txt'. The file 'big.txt' was sourced from Peter Norvig's website [1]. Our spelling correction program also uses portions of Peter Norvig's text correction system [2].

3 Spell Correction

The spelling correction system is based on of the spell check by Peter Norvig. The system starts by reading a sentence from the command line. It first checks if the word is recognized in 'big.txt'. If it is, it is unchanged. However, if the syste, can't find an instance of it, it then generates a large list of every possible spelling combination for the word and stores it into a list. It also keeps track of the previous word, the next word, and the number of times that same combination occurred throughout 'big.txt', stored as (previous, current, next, frequency). The system then looks for any possible matches in the list of data from 'big.txt'. If multiple are found, it uses the most frequent occurrence. If none are found, it leaves the word as is.

3.1 Good Test Case

The system works best on large words. Words like "amazing" are successful almost every time. Since there are less possible real words resulting from one edit and two edits, it also runs quickly. Below is the sentence fragment "teh donut smells amazng". The above text is the input from the user, and the bottom text is the correction the program gave back. The program was able to confirm the words that were already spelt correctly, and correctly identified and replaced the words that were not.

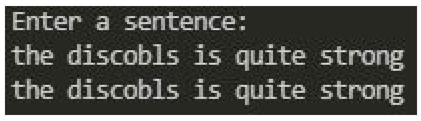


```
Enter a sentence:
teh donut smells amazng
the donut smells amazing
```

Figure 1: A successful spell check case

3.2 Bad Test Case

Since the system was built off of 'big.txt', many arbitrary words aren't understood by the system. For this example, I used the word discobolus; a disc thrower in ancient Greece. The system wasn't able to recognize the word and just left it as is.



```
Enter a sentence:
the discobls is quite strong
the discobls is quite strong
```

Figure 2: An unsuccessful spell check case

4 Word Prediction

We designed our program to take a string input as a command line argument. The program continues to run until the user enters "quit", in which case the program will concur. The input is parsed into a list that can be indexed through. After obtaining the list of words, the previous word and the current word are set, which is the word second to last and the last word respectively. The program will search through the corpus and put all the pairs of previous and next words that have the input current word in the middle. Then the frequency is calculated using the frequency function which in turn calls the occurrence function. This will not only calculate if a pair shows up numerous times but also add that number to the pair as a frequency counter. This counter is used to determine which word to print out to the user. The frequency function will also remove all instances of duplication in order to prevent excess data from staying in the list. As it removes the duplicates, the rest are stored into the final list where all the unique pairs with their frequencies are stored. After this information is gathered and stored, a most frequent function is called that will return the index of the pair located in the final list. From this we print out the user's input line and the program's prediction. The user can either continue another line or chose to quit the program from there.

4.1 Good Test Case

The good test case is in the Word Prediction folder, called "Best Case.txt". Using the line in the file as the input, the user ended up with the next word prediction to be "and". This result can either be perceived has bad or good as the input line could have ended there with a period or continue with the use of "and".

4.2 Bad Test Case

The bad test case in also in the Word Prediction folder, called "Bad Case.txt". Doing the same for the input as in the Good Test Case section, the user ended up with the next word prediction to be "a". This does not fit in the context of the sentence. We were either looking for a period to end the sentence or for a word to continue the sentence, such as "was" which would indicate the sentence to continue. The main issue with the sentence we chose is the lack of variability in the big.txt file, as "frog" only shows up four times in the entire corpus. This does not give a lot of context to the program, so it just gives the user the most frequent next word after "frog".

5 Conclusion

Our systems were successful, but weren't without flaws. The run time on some of the most common words, such as 'the', 'a' or 'it' is extremely long. Trying to replace a misspelling of the word 'the' took over 20 minutes to find a match. The way the system is currently built, it requires multiple scans of 'big.txt'. In addition to searching through 'big.txt', it also has to search subsets of 'big.txt' which causes an exponential run time. In the future, writing a separate program to create all of the lists would be a better route. This way, at run time, it doesn't need to build the same lists over and over. For our predictive text, looking at more words and parts of speech would drastically improve our results. Only looking at the last two words is limiting in the systems ability to understand the context.

6 References

1. https://norvig.com/big.txt

2. https://norvig.com/spell-correct.html