CS459 Mini Project 3

ROSALINA DELWICHE*, Clarkson University, USA
HOLLY ROSSMANN*, Clarkson University, USA

1 INTRODUCTION

For Mini Project 3, our task was to create a modern and unified predictive text system that incorporates both spell correction abilities as well as being able to predict what word a user might type next. Within this program, we must include the ability to read from a file or accept prompts from an input from a body of text. In either case, the system will work with the user to automatically correct misspellings in a body of text, as well as provide a suggestion for the upcoming word based on the current word that a user has typed. Our completed program works as intended, but both has instances of excelling as well as some shortcomings. The following report will walk through the process of the foundation behind each part of our system and how they were implemented as well as how they were unified together for a cohesive program.

2 SPELL CHECKER

2.1 Formulation of Design

When designing the spell checker, we first asked ourselves the question - How are typical errors in spelling made? Without going out and collecting data ourselves, we did some research. One helpful tools we found was Wikipedia's page for Commonly Misspelled English Words [1].

In examining Wikipedia, we were able to find out how most mistakes arise. To start, the word "absence" was commonly misspelled as (1) "absense", (2) "abcense", and (3) "absance". In the first misspelling, one character was off at the end; there was a 's' instead of a 'c'. In the second misspelling, each 's' and 'c' in the word was mistaken for each other. In the third, the first 'e' in the correctly spelled version was written as an 'a'. From this it was clear that given the word "absence", the user is likely to misspell by one or two characters. Given other words such as "acceptable" and "acceptible", this was also the case.

Next, we followed the list for more ways of misspelling. One thing that we noticed was misspellings of the word "achieve" as "acheive". Although at first glance this looks like it is to characters off, which it technically is, there was more than that. More so, the ordering of two neighboring characters were swapped - the 'i' for the 'e' and the 'e' for the 'i'. This was escpecially common in words with the string "ie". Another example was "fiery" being misspelled as "firey", "achieve" as "acheive" and "atheist" as "athiest". In analyzing the further misspellings for "atheist" we noticed one misspelling in particular that stood out.

"Atheist" was also commonly misspelled as "athist". The only element missing here is the 'e' before 'i'. We noticed that other words such as "arctic" were misspelled as "artic", "conscious" as "concious" and many more. It is clear that all these examples were one letter short and one short in length also from the correctly spelled version. This goes the other way, also. Some misspelled words had an extra character. For example "awful" was mistaken for "awfull" ,"harass" mistaken for "harrass" and so on. In many of these cases, there was an extra instance of one character.

In all these instances the misspellings were not obscure, rather, they followed a specific pattern. So, through this analysis, we were able to formulate an answer to our original question, as listed below.

(1) Swapping two neighboring characters
(2) Adding an extra character
(3) Missing a character
(4) One character off, with same length as correctly spelled version
(5) Two characters off, with same length as correctly spelled version

From this, it was clear that the way to correct misspellings was through expecting the itemized list.

2.2 Implementing Design Aspects

We coded using Java (in the IDE Intellij). To start, we imported a text file containing a wide array of English words [2]. The list contains around 470,103 words from the English language. It was important to get a large list for more accurate representation. In addition, we also imported a text file containing common English Names [3] and created a file for the user to put words they do not wish to be corrected. Names and learned words are not spell checked, as there is far greater variability. But nevertheless, they are still important to have to not mistake a word for being incorrect when it is not.

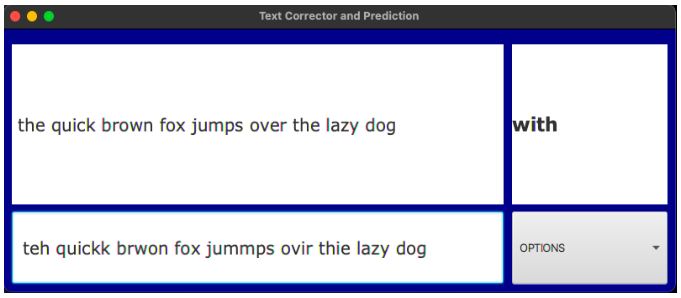
We created a function that takes one word to validate that the word is in fact a word. If not, a list of words were computed on the basis of List (2.1). We choose not to go beyond two characters off, as the list of possibilities would be too large, leading to mistakes. Our initial thought was to create a precedence on how items are commonly misspelled, but we do not know how the user in which way the user is most likely to misspell. Instead, all were computed, until the next step.

In the next step of spell correcting, we came up with a way to narrow down the options. Originally, we read in a Corpus from the Corpus of Contemporary English [4] that contained a word and it's frequency. However, we soon realized that the Corpus only contained around 5,000 words with it's frequency, most of which were not very common. Instead, we based our spell checking on "count_1w.txt" by Peter Norvig. In this text file, it lists some of the most frequent English words, as well as their counts. Our idea was the higher the count, the more frequent the word is.

From this we narrowed down each list to be the most frequent word. Ultimately, the word with the highest count from a compilation of List (2.1) was returned. We further created a function that

Authors' addresses: Rosalina Delwiche, delwicrg@clarkson.edu, Clarkson University, 10 Clarkson Avenue, Potsdam, New York, USA, 13699; Holly Rossmann, rossmah@clarkson.edu, Clarkson University, 10 Clarkson Avenue, Potsdam, New York, USA, 13699.

spell corrects a body of a text as a whole. This can be done in real time or through file.


Text Corrector and Prediction

the quick brown fox jumps over the lazy dog

with

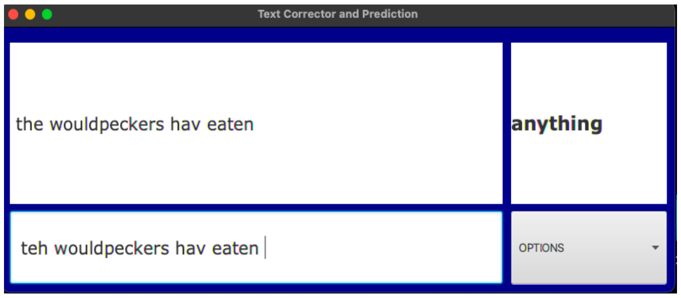
teh quickk brwon fox jummps ovir thie lazy dog

OPTIONS

Fig. 1. Good Test Case - Spell Checker

2.3 Good Test Case

Incorporating the idea of how the may misspell words, we used the example "teh quickk brwon fox jummps ovir thie lazy dog". In this case, the program corrected it to "the quick brown fox jumps over the lazy dog", as shown in Figure 1. Some of the words were already spelled correctly and the program was able to validate that and keep it as is. However, the words that were misspelled were changed according to the Design illustrated above, creating the correct sentence. More specifically, "teh" changed to "the" as the 'e' and 'h' were switched. Next, "quickk" changed to "quick as it had an extra character and "brwon" changed to "brown". Lastly, "ovir" changed to "over" and "thie" changed to "the".


Text Corrector and Prediction

the wouldpeckers hav eaten

anything

teh wouldpeckers hav eaten

OPTIONS

Fig. 2. Bad Test Case - Spell Checker

2.4 Bad Test Case

Long words with over two characters different do not preform well. An example of a bad case would be "teh wouldpeckers hav eaten",as shown in Figure 2. The word "wouldpeckers" was meant to be "woodpeckers". This arises as contains one extra character with several other mistakes. The sub-string between the 'w' and 'd' is "oo" vs. "oul", making that substring one character extra in length, with a wrong character. In our design we did not accommodate for this because a difference of too many characters could obscure a word's meaning. In English, the average length of a word is 4.7 characters meaning there would be an endless array of possibilities if we allowed spell check to operate too many characters off. As a result the misspelled word remains untouched and not corrected. In addition, if the user types "has" as "hav", it will not auto-correct as "hav". This is because "hav" is a recognized abbreviation for "Hepatitis A Virus". This is attributed to "words.txt" being too specific. Although it has an array of common words, there are also lot's of recognized abbreviations, but not so common, such as this example. We could fix this by eliminating abbreviations from "words.txt".

3 TEXT PREDICTION

3.1 Implementation

To implement the Text Prediction aspect of our program, we first wanted to gather a good basis for what words often come after a specific word. To do this, we used Peter Norvig's big.txt document [6], that contains data from Project Gutenberg, Wiktionary, and British National Corpus. This document contains a large amount of written text, and we believed that by utilizing this document in particular we would be able to gather a good understanding on word frequency.

To begin, we created a function to read in all of the words from big.txt and enter it into a vector, with each entry being a single word. We did this using a scanner, where one line would be read in at a time. For each line, it would then be broken down by spaces, special characters would be removed, and the text would be converted to all lowercase. This ensures that we have nothing but the raw text of each word in the vector, which we can then work on using to count for word frequency.

The next function we created was to calculate this next word frequency. It takes in the current word (the most recent word that the user has typed), and we use this current word to sort through the vector of all words from big.txt. Whenever an instance was found of the current word in the vector, it would then select the next word in that vector and insert that into a new vector. This vector then became accumulated of every single "next word" that came after the current word. From there, we created a hash map, and cycled through the next words vector to count up the total of each word's appearances. After this was done, the map would then consist of each next word that appeared after the desired current word, and the amount of times it appeared. This allowed us to gauge the frequency in which certain words appeared after the target word, which we can then use to aid in our predictions.

After that, we the moved on to a function that then calculated what exactly the prediction should be. Here, we created a system that looks through the hash map for the highest element, and then pulls the corresponding key (word) from that map. This was done so that from all the possible next words, we would end up displaying one of the most common ones, as that has a higher percentage of being the word that the user desired. If the user inputs a particular word or abbreviation (as the abbreviation "hav" was discussed earlier) that is not present in big.txt, then the program is able to handle this, and would return an empty string to demonstrate that there is currently no prediction for that word.

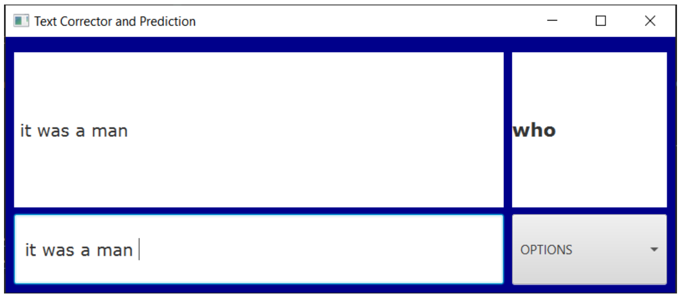


Fig. 3. Good Test Case - Predictor

3.2 Good Test Case

Many cases work well with our current text prediction. Given a particular word, the program will successfully retrieve the most common word that would normally come next. This can be seen in the test case shown in Figure 3. The user was currently typing a sentence "It was a man..." and as shown in the right hand display, the next word is predicted to be "who". This makes sense grammatically, as the user could be typing the sentence "It was a man who talked to me earlier today." We consider this a successful test case, as the prediction is relevant to the current sentence, makes grammatical sense, and can be added onto to create a full sentence.

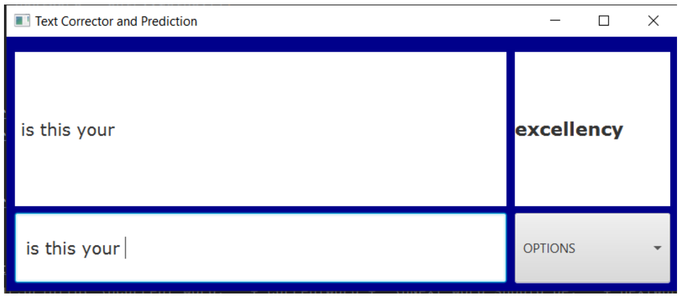
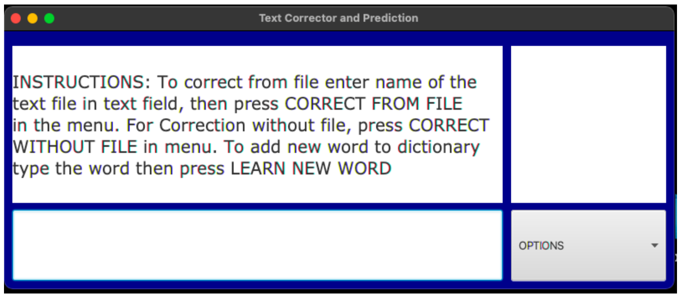


Fig. 4. Bad Test Case - Predictor

3.3 Bad Test Case

While there are many cases where our predictor works as intended, there are still some areas where improvement could be made. In this sense, one bad test case can be seen in Figure 4, where the user is typing "Is this your...", and the next word prediction is shown to be "excellency." While this may technically make grammatical sense, the sentence more likely could have been "Is this your pen" or some other object, as opposed a reference to someone's title. The big.txt document does involve many old and classic texts, which could attribute to "excellency" being the most popular word to come after "your". However, in modern years most people do not speak in this way, and excellency is not a part of many people's current vocabularies. Therefore, even though this may still make sense, it is not a good indication of what most people would wish to type next in this circumstance.



INSTRUCTIONS: To correct from file enter name of the text file in text field, then press CORRECT FROM FILE in the menu. For Correction without file, press CORRECT WITHOUT FILE in menu. To add new word to dictionary type the word then press LEARN NEW WORD

Fig. 5. Text Predictor and Spell Checker - Opening Prompt

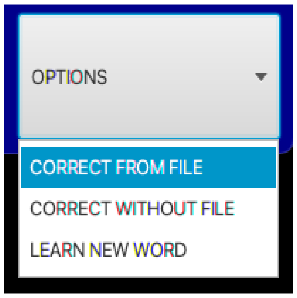


Fig. 6. Menu Options

4 UNIFICATION OF SPELL CHECK AND TEXT PREDICTION

To unify both the spell checker and text prediction aspects of our program, we decided to make a GUI that would be able to be used by the user. When this program is first started, it gives the instructions as shown in Figure 5. The instructions explain there is 3 modes to the GUI - correction and prediction using a file, or without a file and instead in real time, as well as an option to add a word to your personal dictionary. This drop down menu is shown in Figure 6.

For the "Correct with File" option, the user would then enter the name of the text file in the lower left text-field and press the corresponding menu options. If the file is successfully found, the text is read in and corrected in it's entirety with the spell checker. The corrected text shows up in the upper left box and the suggestion for the next word in the box on the right hand side. This word suggestion would come from the last word in the file. In addition, the corrected text is written back into the file it received it from.

For the "Correct without File" option, the user will select the corresponding menu option, and it will automatically begin spell correction and prediction. In both the cases of "Correct with File" and "Correct without File", the user can press the down arrow to accept the word suggestion. An example is given in Figure 7.Here, the upper left box contains the spell checked version - "at the", while the lower left input box still contains the non-spell checked - "at teh". We believed this is important for the user to see what they originally were typing, in case they do not agree with the spell correction. This is due to the fact that while wrong spellings, such as "teh", may be a clear issue but others, such as the spelling of an uncommon name, may not be, and we want the user to be able to type out their name without constantly having this change on them in their input box. In addition, the word suggestion "same" is shown in the upper right box, otherwise referred to as the "suggestion box".

This unification incorporates both the text correction and text prediction, and smoothly works to bring the two together for a correctly working program. Each portion of the program has its designated section, and this is to ensure that the user does not feel forced to accept the spelled word or next word predictions. Both the with and without file options operate as intended, and they both provide a pleasant experience for the user.

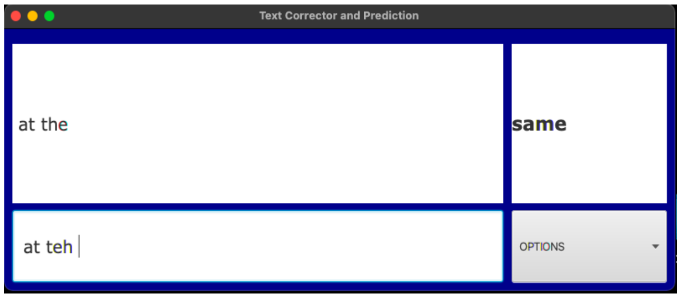


Fig. 7. Word Predictor and Spell Checker Example

5 FUTURE WORK

5.1 Spell Checker

In our future work, we would like to add more features to the spell checker. More specifically, explore different way the user may make mistakes and examine it more in the context of forms of speech. For example, narrow down our list based on whether the spell checked word comes after a subject, verb or adjective, thus being more viable.

5.2 Text Predictor

In the future, we would ideally also like to make improvements to the text predictor. One important improvement would be to not just always display the most common next word, but instead cycle through the top common words in the map for more variety. This would help to eliminate instances where very common words are repeatedly suggested in multiple scenarios. Another important addition would be to take into account more part of a sentence than just the current word. As the program works, it currently only examines one word to formulate what the most popular next word should be. However, the system would be more accurate if it was able to reflect on the entirety of the sentence in making the calculations. This could also help with choosing a word with the correct part of speech, or helping with typing out common phrases that are multiple words long. Another possible item to address would be allowing the program to learn from a user's text habits to create more personalized suggestions based off of their past interactions. Our program works as intended and performs well, but as always there are very many new paths, directions, and improvements that could be made to create a more intricate and accurate system.

6 BIBLIOGRAPHY

1 https://en.wikipedia.org/wiki/Commonly_misspelled_English_words
2 https://www.mit.edu/~ecprice/wordlist.10000
3 https://www.usna.edu/Users/cs/roche/courses/s15si335/proj1/files.php%3Ff=names.txt&downloadcode=yes
4 $https://www.wordfrequency.info/samples/lemmas_60k_words.txt
5 http://norvig.com/ngrams/count_1w.txt
6 https://norvig.com/big.txt