Final Project Report: Spell Checker

Frank Anderson

# Introduction

I developed a spell checker to read plain text files and provide the user with suggestions of correctly spelled words which are similar to the misspelled word. The program writes the chosen suggestion instead of the misspelled word to a new file.

To find misspelled words, the program tries to read the word from a hash map of words generated from a file containing a list of words. If the word can be read, the program moves to the next word. If not, the program manipulates the word to check if it’s a plural or possessive form and checks the original form of the word. If the word is still not in the dictionary after these tests, the app generates a list of suggestions.

To create the list of suggestions, the program manipulates the given word in a series of ways and checks if any of the manipulations are words. If the manipulations are words, they are stored in a list and the results are presented to the user. To manipulate the words: the program tries replacing each letter in the word with other letters; inserting other letters between each set of letters; swapping each set of adjacent letters; and removing each letter once. The program also has the ability to combine manipulations in sequence, so after running each test on its own it combines each test with each of the others and itself to try and generate even more options. This method is far more effective than my original method, checking the similarity of every word in the dictionary with the given string, because it only requires checking the validity of a few words instead of every word in the dictionary.

The results are entirely dependent on the dictionary used and the number of tests run. I spent a lot of time at the start of the project testing paragraphs from a variety of sources on a variety of dictionaries I found. After testing, I decided it would be best to combine dictionaries to be able to search more words. I used the “words.txt” list from <https://github.com/dwyl/english-words> and “contractions.txt” from <https://gist.github.com/J3RN/ed7b420a6ea1d5bd6d06>. The results also depend on how many other words are more similar to what you were trying to type, so more obscure words receive fewer suggestions than shorter, more common words because there are more short, common words than long words.

# Application Design

The system is comprised of a SpellCheckApp class, Suggester class, and Dictionary class, and PathManager class. The SpellCheckApp class owns instances of the Suggester and PathManager. The Suggester owns an instance of Dictionary. Initializing the SpellCheckApp starts the program.

The SpellCheck class creates a Suggester, which creates a Dictionary. The Dictionary reads over a series of word lists and stores the words in a hash map. Then the SpellChecker creates a PathaManager. On initialization, the PathManager asks the user for a file name and checks if the file name is valid. Then it formats the file name so that there’s a file to read from and a file to write to. Then, the SpellCheck class opens the files to read from and write to and begins reading the file character by character.

If the character is not whitespace or punctuation, it adds the character to a buffer. If the character is whitespace or punctuation, it processes the string in the buffer. To process the string, the word has special characters and whitespace stripped and is then passed to the Dictionary to check if it’s a word. If it’s a word, the word is output to the console and file and the buffer is cleared. If it’s not a word, the program checks if the word is a plural or possessive form and then checks the original form of the word. If those forms are also not valid words, the program generates suggestions.

Suggestions are generated by the Suggester class using methods described in the introduction. The SpellCheckApp class then lets the user pick one of the suggestions, and writes that to the console and output file.

# Class Diagram

UML Class Diagram of each class and its functions.

Diagram

Description automatically generated

# GitHub

<https://github.com/fraander/spellchecker-eece2140project>

# Instructions

1. Download the code from the GitHub page and navigate to the project-files folder.
2. Create a plain text file in .txt format to be read by the program. Save it in the same location as the main.py and other program files.
3. Run the main.py script, type in the name of your .txt file including the file extension.
4. Follow the on-screen directions, using the number keys to indicate your choices and enter to confirm.
5. When the program has finished running open the “edited+[your file name]” file to see a file with your results. Your original file is left in place so that you can compare the results and keep a version history if you’d like.

# Libraries and Tools

To build the program, I used two dictionary lists.

* “words.txt” from: <https://github.com/dwyl/english-words>
* “contractions.txt” from: <https://gist.github.com/J3RN/ed7b420a6ea1d5bd6d06>

# Lessons Learned

The biggest lesson I learned from this project was that the first solution is not the best solution. At first, I tried comparing the word to all the other words in the dictionary and returning the most similar strings, which is highly inefficient. After presenting, it was made apparent to me that most spell checkers do not use this method, because of its inefficiency. Instead, they use the method I settled on to generate a few words to check that are similar. The new method I used produced better results too, much to my chagrin!

My favorite part of the project was being able to use my object-oriented programming skills. Interfacing the different classes and organizing my many methods helped me think through the different problems, and I enjoyed componentizing each operating into more generic methods. I would recommend future EECE 2140 students get comfortable working with OOP because it will make them better, more deliberate programmers.

This project also helped me improve my Git skills. At one point I tried branching features to try a new technique, something I should have done even more. It was also much easier to try making major changes to the program with the knowledge a previous working version was safe and accessible. My hope is that future EECE 2140 students learn Git earlier in the semester and make use of it on Problem Sets as well.

Finally, I learned how to componentize a large, complex problem into smaller, solvable pieces at a scale I had not before. Breaking down a file into words, generating suggestions from a string, and handling numerous inputs and outputs were all smaller problems I had to solve. It was empowering to feel comfortable attacking each of these parts and then combining them into a useful program after taking this course. I hope future EECE 2140 students are able to feel the same way.