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 reads a dictionary from a plain text file. Then, as the program iterates through the file of the user’s choice word by word, it compares each word to those in the dictionary text file. If they are present the program writes the word to the output file and continues to the next word. If the word is not in the dictionary, the program tries a possessive and plural version, and then asks the user to choose from a list of suggestions.

To create the list of suggestions, the program uses the Fuzzy Wuzzy library to compare the given word to all the words in the dictionary. Fuzzy Wuzzy produces a ratio which is a value from 0 to 100, 100 being the words are the same. I then have the program display the three most similar values to the user to choose from. Alternatively, the user can choose to write something of their own or to use the original word. I tried writing my own version of fuzzy search where I gave each word a difference score based on how many letters were different from the original word, but the method returned too many results.

I tried to filter options where the only different letters were only those which were close to the letter which was originally tried (for example: “sord” and “qord” but not “bord” would be valid replacements for “word” because “s” and “q” are adjacent to “w” but “b” is not) but this limited the results too much. I also noticed many errors were based on not knowing how to spell words, not just typing the wrong letters. Hence, I decided to use Fuzzy Wuzzy since its results were so much more effective and I was running low on time.

The results are entirely dependent on the dictionary used, so 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 more accurate suggestions because very few words are similar to them.

# Application Design

The system is comprised of a main class, GrammarChecker, and two helper classes, Dictionary and FileManager.

GrammarChecker is the class that iterates over each word in the file and controls the logic for what to do if words are spelled correctly or not, and what to do when the user wants to use a different word. It contains an instance of the FileManager and the Dictionary. Most of this logic is handled in the initializer because it makes using the checker very user friendly; all the user must do is initialize a new GrammarChecker instance. The GrammarChecker has helper methods to print the options found, ask the user what choice they want to select, and ask the user for a custom word.

Dictionary is the class which handles reading the dictionary files and the convenience methods for strings. It stores the list of words as an instance variable after reading them in during the initializer. The clean\_input function removes special characters from strings to make them searchable. The word\_in function checks if a given word is in the dictionary or if its plural or possessive forms are. Finally, the find\_similar function finds words similar to the given word. The GrammarChecker class has an instance of the Dictionary and accesses it as needed.

Finally, the FileManager class handles the file path for the file to read and write to. When the FileManager is initialized, it runs validate\_filepath where it asks the user for a file name and checks that the file path is valid. When an instance of GrammarChecker is created, an instance of FileManager is created and the path is validated before the rest of GrammarChecker can be run.

# Class Diagram

UML Class Diagram of each class and its functions. Initializers are only shown if they have helper functions as well.

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 and the Fuzzy Wuzzy library.

* “words.txt” from: <https://github.com/dwyl/english-words>
* “contractions.txt” from: <https://gist.github.com/J3RN/ed7b420a6ea1d5bd6d06>
* FuzzyWuzzy documentation: <https://pypi.org/project/fuzzywuzzy/>

# Lessons Learned

This project made me realize just how much more complicated the rules of language than the rules of nearly anything else. English has so many different words are rules for the contexts they can be used. One instance is the sheer number punctuation marks I had to account for—for example: quotation marks can be directional, there aren’t just single and double quotes! Because of this, most of the time I put into the project was learning to clean and sanitize strings so that they could be sent through the program, not into writing the core logic of the program itself. I was excited to write a spelling and grammar checker, but I had to limit the scope to just a simple spell checker because there is so much complexity in the English language. If I had put a little more time into testing some ideas before submitting my proposal, I think I would have picked something more simple and able to be broken down into core parts. Instead, this project has one big part.

If I had more time, I would have loved to better understand methods that operating system spell checkers use because I think they would have provided a much-needed source of inspiration for a longer-term project. I did learn that Linux uses a system similar to mine for simple spelling checks (I tried using their dictionary because Linux is open source) but I also learned they break words down further to test tenses and plurality more efficiently without having to store every version of every word. This would have been interesting to learn more about and to write my own version of.

I would advise future EECE 2140 students to break down the problems they are thinking of solving into parts before they start coding. I broke down the parts of a spell checker but didn’t break down the actual mechanics of checking spelling or suggesting words and very much paid the price for that. Because of this, I might also suggest they choose problems with known scope such as games with defined rules because it lets you spend more time on the code than on just being able to solve the problem in the first place. That said, creating a real-world program is very fun and I would also recommend picking something you are passionate about.

Overall, this project was enjoyable to work on and I learned a lot about writing larger, maintainable software. It would have been fun to use GitHub more throughout the semester, since I did make use of commit history and branching on one occasion to figure out the solution to a regression in my program’s efficiency and that would have been helpful on the problem sets this year too.