User-Focused Autocorrection Using AI Techniques

Dillan Zurowski

200431334

ENSE 480 - Dr. Christine Chan

Table of Contents

Table of Contents......1
1. Introduction......2
2. Knowledge and data representation......2
3. Approach, Techniques and Algorithms......4
4. Structural Diagram......6
5. How to Use Modules......8
6. Sample Sessions......9
7. Sample Listing......10
7.1 The 4 Edit Functions: DeleteLetter, SwitchLetter, ReplaceLetter, InsertLetter......10
7.2 Generation of Edits: edit_one_letter, double_edit......11
7.3 Create List of Suggestions: get_corrections......12
7.4 Create list using surrounding words......12
8. Discussion......13
9. Conclusion......13
10. Future Work......14
References......14

1. Introduction

In today’s world, typing is the main source of documentation,whether it is writing an important research report or sending a friend a text message. As we all know, making spelling mistakes are very common and can often be troublesome. Personally, I’ve been more frustrated with the autocorrect changing my correctly spelt word to something that I’ve never typed and would never type. This is one of the reasons that I decided to create a program that uses AI concepts to correct the user input by focusing on the user and their preferred words. Using concepts based on the NLP(Natural Language Processing), which is a branch of AI that attempts to give the system knowledge about human language, I attempted to give this program the skills needed to be able to understand the user’s habitual language and adjust to them over time. I decided to approach this problem by creating a database of 3000 words that each contain a probability which is adjusted over time and a list of potential surrounding words that have been previously entered. In the end, I wanted the program to detect when a word was misspelled, correct the word using AI techniques, and return the corrected sentence back to the user.

2. Knowledge and data representation

Input Data: chose to represent the inputted data as a list of strings, this way I could iterate through the list while also keeping track of parent and child nodes. The program ignores all special characters and capitals as they do not matter for spelling.

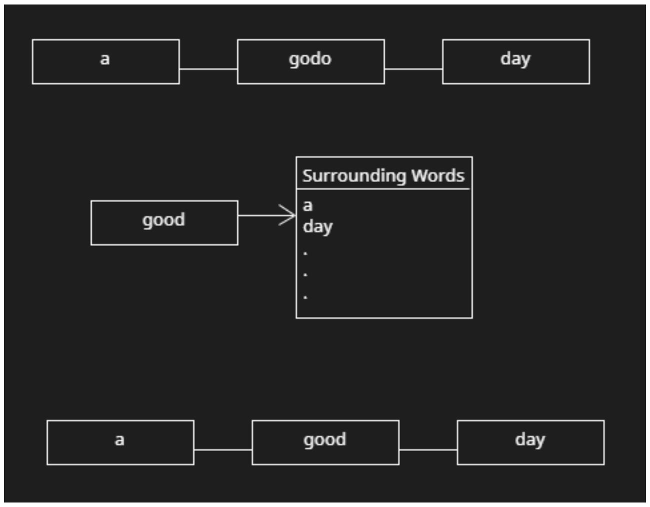


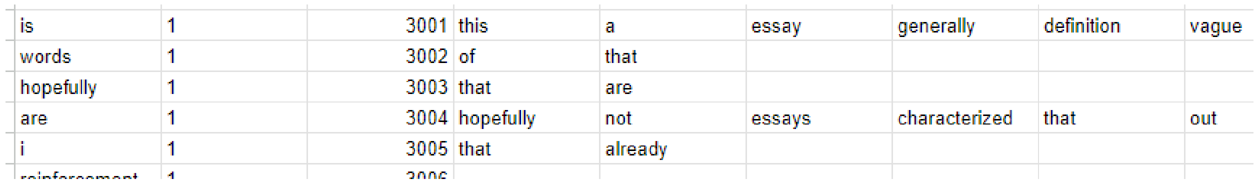
Figure 2.1 - Input data representation

Database:

1. First, I used a txt file that was stored in my github which could be called from google colab. This worked for the storage of the words however, I knew in order to apply AI techniques, I would need to be able to write to the storage and dynamically add heuristics to each word.
2. Second, I used a linear structure in Microsoft Excel where one column was a row of words and the next column was its corresponding probability. My goal at this stage was to dynamically increase the probability of a word each time an incorrect word was changed to said word. Here, I can apply best-first search by sorting the words by highest probability and iterating through those words first. I wanted to use a different method of correction and increase the accuracy of the correction by checking the surrounding words. I also needed to add a column for the index so I can find the index of a word.
3. Finally, I added to the excel sheet so that each row that contains a word will have a list of words that appear around the word. I can use this list of surrounding

words to increase the probability of the word when the incorrect word is surrounded by one of the stored surrounding words.

ex)



is	1	3001	this	a	essay	generally	definition	vague
words	1	3002	of	that				
hopefully	1	3003	that	are				
are	1	3004	hopefully	not	essays	characterized	that	out
i	1	3005	that	already				
reinforcement	1	3006						

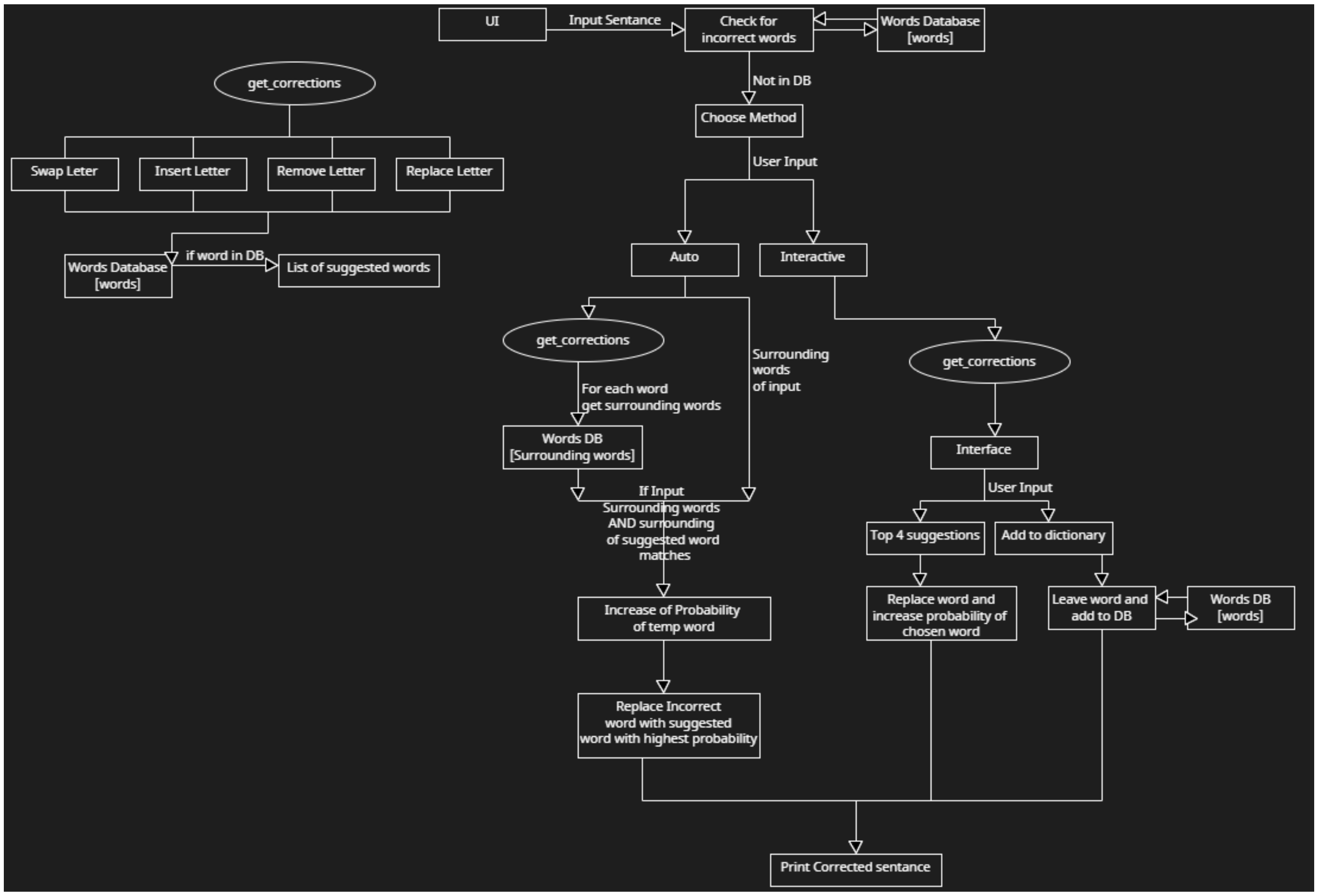
Figure 2.2 - Example of the database from Google Sheets

3. Approach, Techniques and Algorithms

- Since words don't directly relate to each other, creating a tree directly was too difficult which is why I used the heuristics (probability) to determine which words to check first when creating the list of potential solutions. This is inspired off of a blind search where nothing is known about the data, however sorted by probability. Other search algorithms would not apply as there is no relation to each word.
- Probability
    - Option 1: Base probability off of chance of the new word randomly being selected
    - Option 2: Base probability off of how many changes needs to be made to get to a solution
    - Used: Start probability of all words at 1, increase each time the user uses that word as the replacement.

- Edit the incorrect word using 4 algorithms
  - INSERT - Every letter in the alphabet is inserted between every letter in the current word. This catches cases where the user accidentally missed a letter when typing.
  - DELETE - Returns a list of words where each letter has been taken out. This case is for when the user types in a letter by accident.
  - SWAP - Swaps two adjacent letters to catch the case where a user types two letters in the wrong order. An alternative approach could be to swap a letter further than its immediate neighbor.
  - REPLACE - Changes each letter in the current word to every possible letter in the alphabet. This function is the most likely to occur and missed by the user.
  - These four algorithms are were compared with the pip installation “pyspellchecker” and got very similar results. This means these functions produce accurate results.
- A relation can be created with the words around each word as the program compares the words before and after the current word with the list of surrounding words of each returned potential solution. This technique is used to automatically correct the entire sentence without user intervention. This method also dynamically adds the surrounding words to the list if they are not there already.
- Naive Method was used to check if a word is in the database of words. This is a python function that iterates through the list and returns true if found.

4. Structural Diagram



The user input is iterated through word by word, checking if it is a word in the dictionary. If the word is not in the dictionary, the user is then prompted to enter in a method of correction. Both methods generate a list of potential words based off of the current word.

Get_corrections: this function returns a list of potential words. A misspelled word is imputed into this function and the word gets passed to the 4 edit functions: insert, replace, delete, and swap. These functions do as they suggest, insert a letter between each letter in the word, replace each letter, delete a letter, and swap two adjacent letters. Each of these edit functions return a list of words. The get_corrections function takes the combined list of all 4 functions and runs them again for each of the generated words. This produces a very large list of edited words which can be reduced by only returning the words that exist in the dictionary. These words will already have surrounding words associated with them which will be used in the auto-complete method.

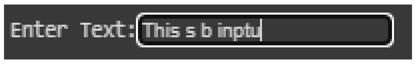
In both methods, the get_corrections function is called and a list of potential words and their stored surrounding words are noted.

If the user chooses to correct all of the words at once, the system will iterate through the list of temporary suggestions from the get_corrections function, checking if the surrounding words of the current word exist within the list of stored surrounding words for each temporary word. If the surrounding words exist in both, the score of the temporary word goes up by 5 for each matching word, adding to the current probability of that word. After iterating through each temporary word, the word that has the highest score will automatically replace the current incorrect word. This will repeat automatically for the entire input.

On the other hand, if the user chooses the user-intervention method. Then the system will display the top four generated suggestions with the highest scores. They then have the option to select one of them which will both replace the incorrect word with that of their selection, as well as increase the probability of that word by 1. Another possibility is that they select the option to add the word to the dictionary which will add a new row to the database containing the row and a default probability of 1. This will repeat for all misspelled words.

Both methods will eventually print the final corrected sentence.

5. How to Use Modules

1. Open code in Google Colab
2. Open CSV in Google sheets
3. Run the code, allowing the system to connect to your google drive to access the csv file.
4. Enter in any string
    a. 

```
Enter Text: This s b inptu
```

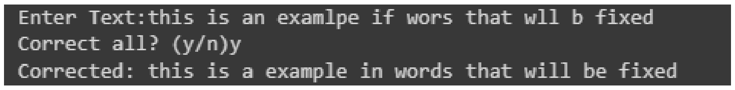
5. Choose a method of correction
    a. Correct all - Enter “y” in the input to correct all words at once. Uses the 4 edit functions to create a list of possible words and compares the surrounding words with the heuristics of the temporary solutions to iterate through the whole sentence and generate the most accurate result.
    b. User selection - Enter “n” in the input to give the user a list of highest probability words and allow the user to select one of them. When the user selects one of the options, the probability of that word will increase. The user also has the option to add the “incorrect” word to the database, where the base probability will be 1.

    i. Select one of the suggested words by entering a number from 1-4. This will replace the current word with that selected word.
    ii. Add the word to the dictionary by entering the number 0. This adds a new word to the google sheets document.

6. View final corrected sentence.

6. Sample Sessions

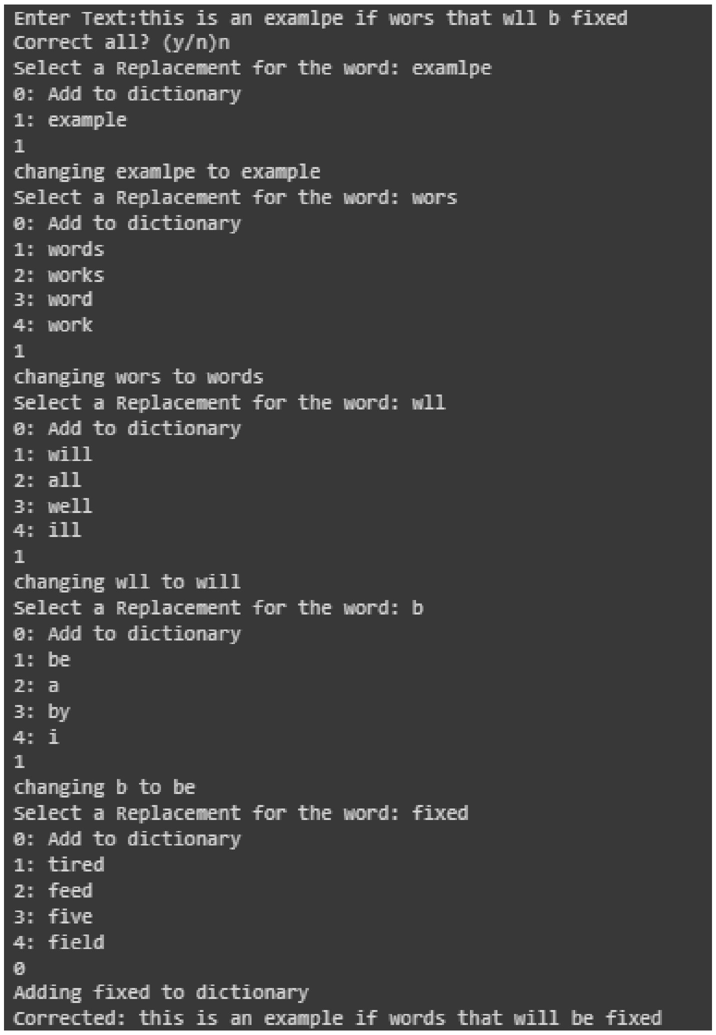
1. Autocorrect: iterates through the input and sends incorrect words to the “correct_using_neighbors” function.



```
Enter Text:this is an examlpe if wors that wll b fixed
Correct all? (y/n)y
Corrected: this is a example in words that will be fixed
```

Figure 6.1 - Sample of Correct All

2. User Selection - iterate through, allowing user to select each time



```
Enter Text:this is an examlpe if wors that wll b fixed
Correct all? (y/n)n
Select a Replacement for the word: examlpe
0: Add to dictionary
1: example
1
changing examlpe to example
Select a Replacement for the word: wors
0: Add to dictionary
1: words
2: works
3: word
4: work
1
changing wors to words
Select a Replacement for the word: wll
0: Add to dictionary
1: will
2: all
3: well
4: ill
1
changing wll to will
Select a Replacement for the word: b
0: Add to dictionary
1: be
2: a
3: by
4: i
1
changing b to be
Select a Replacement for the word: fixed
0: Add to dictionary
1: tired
2: feed
3: five
4: field
0
Adding fixed to dictionary
Corrected: this is an example if words that will be fixed
```

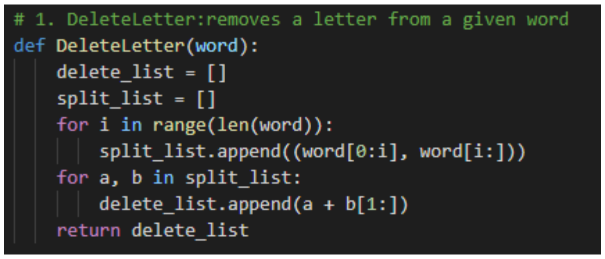
Figure 6.2 - Sample of user selection method

7. Sample Listing

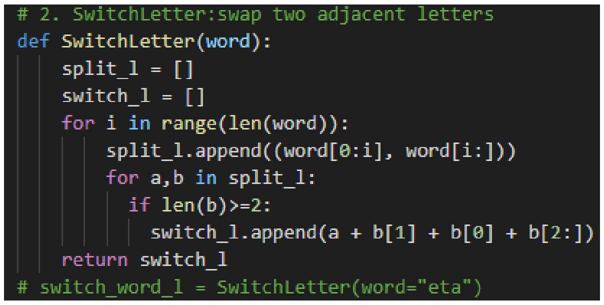
7.1 The 4 Edit Functions: DeleteLetter, SwitchLetter, ReplaceLetter, InsertLetter

Input parameters: String - misspelled word to be edited

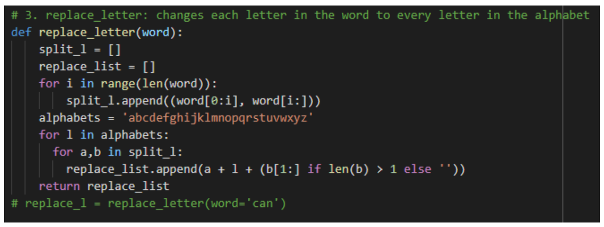
Output parameter: List of Strings - List of potential words



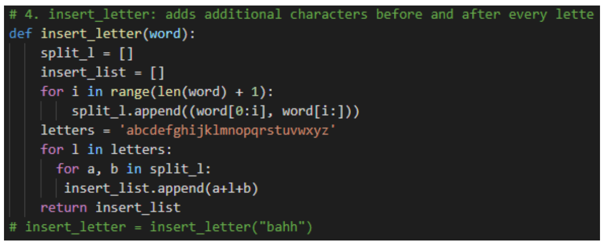
```python
# 1. DeleteLetter:removes a letter from a given word
def DeleteLetter(word):
    delete_list = []
    split_list = []
    for i in range(len(word)):
        split_list.append((word[0:i], word[i:]))
    for a, b in split_list:
        delete_list.append(a + b[1:])
    return delete_list
```



```python
# 2. SwitchLetter:swap two adjacent letters
def SwitchLetter(word):
    split_l = []
    switch_l = []
    for i in range(len(word)):
        split_l.append((word[0:i], word[i:]))
        for a,b in split_l:
          if len(b)>=2:
            switch_l.append(a + b[1] + b[0] + b[2:])
    return switch_l
# switch_word_l = SwitchLetter(word="eta")
```



```python
# 3. replace_letter: changes each letter in the word to every letter in the alphabet
def replace_letter(word):
    split_l = []
    replace_list = []
    for i in range(len(word)):
        split_l.append((word[0:i], word[i:]))
    alphabets = 'abcdefghijklmnopqrstuvwxyz'
    for l in alphabets:
      for a,b in split_l:
        replace_list.append(a + l + (b[1:] if len(b) > 1 else ''))
    return replace_list
# replace_l = replace_letter(word='can')
```

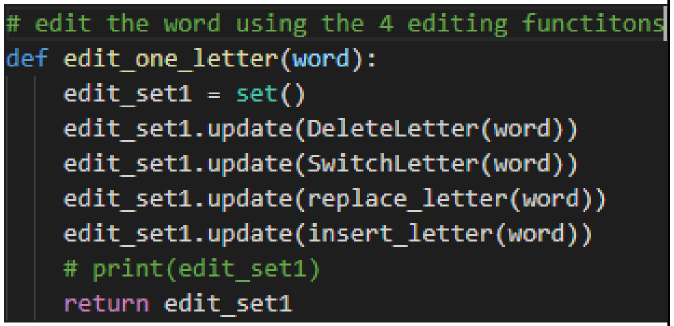


```python
# 4. insert_letter: adds additional characters before and after every lette
def insert_letter(word):
    split_l = []
    insert_list = []
    for i in range(len(word) + 1):
        split_l.append((word[0:i], word[i:]))
    letters = 'abcdefghijklmnopqrstuvwxyz'
    for l in letters:
      for a, b in split_l:
       insert_list.append(a+l+b)
    return insert_list
# insert_letter = insert_letter("bahh")
```

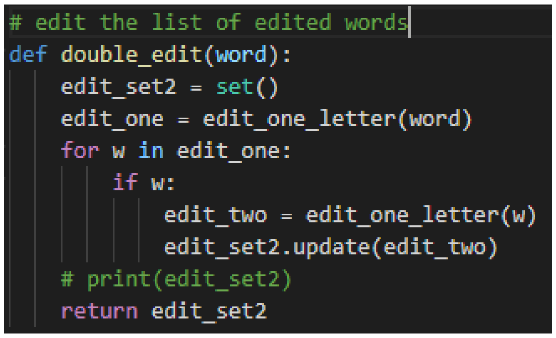
7.2 Generation of Edits: edit_one_letter, double_edit

Input parameter: String - one word to use as a base to generate a new set of words

Output parameter: list of strings - a list of potential words which may or may not be in the dictionary



```python
# edit the word using the 4 editing functitons
def edit_one_letter(word):
    edit_set1 = set()
    edit_set1.update(DeleteLetter(word))
    edit_set1.update(SwitchLetter(word))
    edit_set1.update(replace_letter(word))
    edit_set1.update(insert_letter(word))
    # print(edit_set1)
    return edit_set1
```

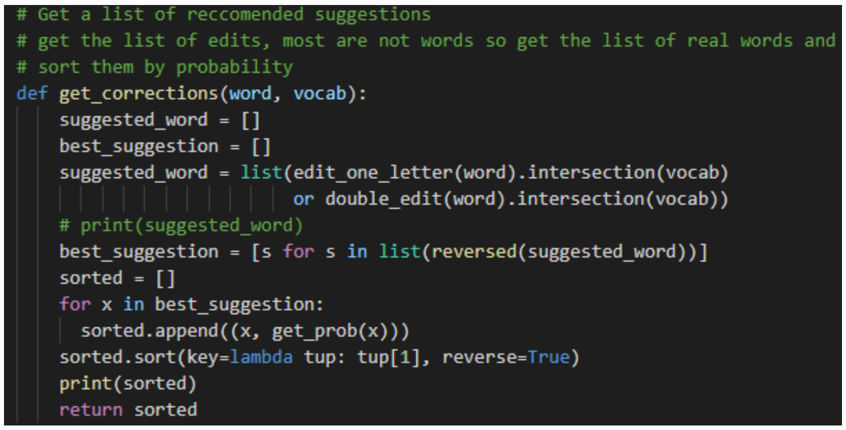


```python
# edit the list of edited words
def double_edit(word):
    edit_set2 = set()
    edit_one = edit_one_letter(word)
    for w in edit_one:
        if w:
            edit_two = edit_one_letter(w)
            edit_set2.update(edit_two)
    # print(edit_set2)
    return edit_set2
```

7.3 **Create List of Suggestions**: get_corrections

Input parameters: String, set - the word to be edited and the set of words in the database

Output parameters: List of strings - list of words sorted by probability

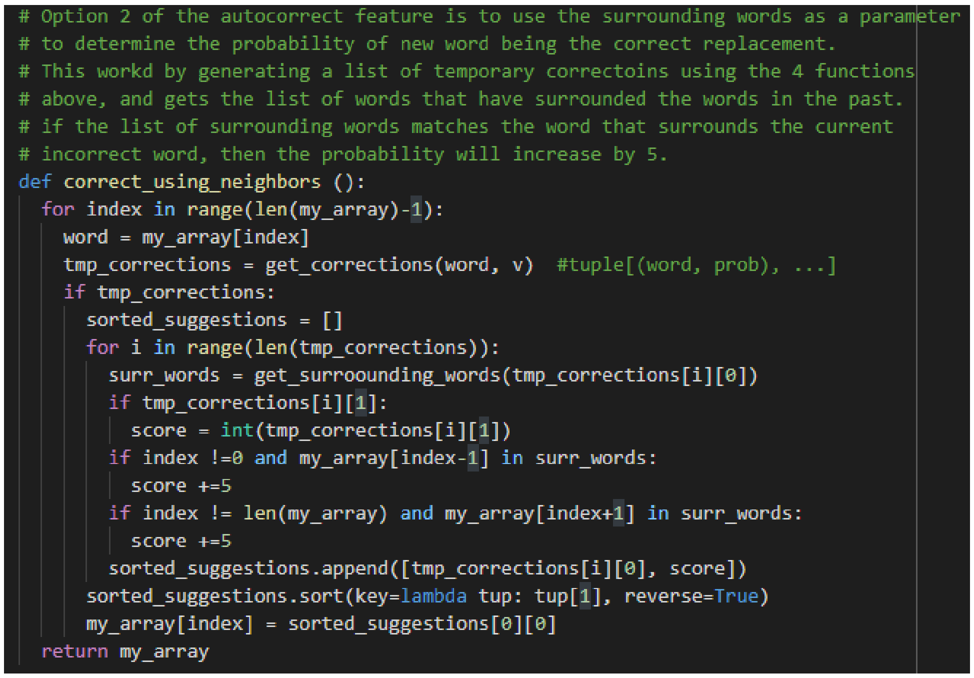


```python
# Get a list of reccomended suggestions
# get the list of edits, most are not words so get the list of real words and
# sort them by probability
def get_corrections(word, vocab):
    suggested_word = []
    best_suggestion = []
    suggested_word = list(edit_one_letter(word).intersection(vocab)
                          or double_edit(word).intersection(vocab))
    # print(suggested_word)
    best_suggestion = [s for s in list(reversed(suggested_word))]
    sorted = []
    for x in best_suggestion:
        sorted.append((x, get_prob(x)))
    sorted.sort(key=lambda tup: tup[1], reverse=True)
    print(sorted)
    return sorted
```

7.4 Create list using surrounding words

Input parameters: my_array - inputted list of words

Output parameters: my_array - fixed array of words



```python
# Option 2 of the autocorrect feature is to use the surrounding words as a parameter
# to determine the probability of new word being the correct replacement.
# This workd by generating a list of temporary correctoins using the 4 functions
# above, and gets the list of words that have surrounded the words in the past.
# if the list of surrounding words matches the word that surrounds the current
# incorrect word, then the probability will increase by 5.
def correct_using_neighbors ():
    for index in range(len(my_array)-1):
        word = my_array[index]
        tmp_corrections = get_corrections(word, v)  #tuple[(word, prob), ...]
        if tmp_corrections:
            sorted_suggestions = []
            for i in range(len(tmp_corrections)):
                surr_words = get_surroounding_words(tmp_corrections[i][0])
                if tmp_corrections[i][1]:
                    score = int(tmp_corrections[i][1])
                if index !=0 and my_array[index-1] in surr_words:
                    score +=5
                if index != len(my_array) and my_array[index+1] in surr_words:
                    score +=5
                sorted_suggestions.append([tmp_corrections[i][0], score])
            sorted_suggestions.sort(key=lambda tup: tup[1], reverse=True)
            my_array[index] = sorted_suggestions[0][0]
    return my_array
```

8. Discussion

The goal of this project was to be more user-focused and dynamic then a typical autocorrect function. My program achieves this goal as the database gets more accurate the more the user uses the system and inputs their preferred choices of words. Some pros of this program include its dynamic nature, its ability to learn user preference, new words, and new heuristics over time, and its ability to produce more accurate results over time. Cons include the dataset not being full due to the amount of time it takes to read and write to the google sheets database using the API. I used the training function that I created and it took almost 30 minutes to write a paragraph of new words and the surrounding words due to it checking if the word exists, finding the word in the database, then writing to an available spot in the list of surrounding words. The accuracy of the program would be greatly increased if the database was full.

9. Conclusion

In conclusion, the program I created is very user-focused and implements a concept of constant learning and improvement. In the future, I would like to explore ML concepts and learn more about how to train my dataset in a more efficient way.
I wanted to avoid using too many libraries to challenge myself in solving different problems and apply concepts in a new language. I am satisfied with the way my project turned out and have learned new concepts that I can apply in the future.

10. Future Work

If I had time and resources, I would find a better way to communicate to the database containing the words and heuristics. Currently, the API used to communicate between the google sheets and the google colaboratory takes too long to read and write. For example, the training function to write to the database takes several minutes to populate the database with only a paragraph of new information. I would research a different method to improve this process however for the purposes of this project, the time scale was not an issue. One solution that I would have explored first would be to implement a SQL database as I found plugins similar to the google sheets plugin. This would make the organization of heuristics easier as well. In hindsight, I also should have implemented a better user interface and created functions to decrease the repeated tasks to decrease the amount of repeated code. Another way that I would have extended my project is if I could find a way to detect sentence structure such as nouns and verbs to further increase the accuracy of the suggestions. This would be easy due to the system already reading the surrounding words.

References

Kadlaskar, Amruta. “Autocorrect Feature Using NLP in Python: What Is Autocorrect.” Analytics

Vidhya, 8 Nov. 2021, https://www.analyticsvidhya.com/blog/2021/11/autocorrect-feature-using-nlp-in-python/#:~:text=Now%20let%20us%20get%20started,as%20f%3A%20file_name_data%20%3D%20f

“Building an Autocorrect Feature Using NLP with Python.” Section, 4 Dec. 2021, https://www.section.io/engineering-education/building-autocorrect-feature-using-nlp-with-python/

“English-Words.” PyPI, 5 Jan. 2023, https://pypi.org/project/english-words/#:~:text=To%20add%20a%20word%20list,run%20the%20script%20process_raw_data.py%20

Python - Spelling Check. (n.d.). Tutorialspoint. Retrieved April 8, 2023, from https://www.tutorialspoint.com/python_text_processing/python_spelling_check.htm