AutoCorrect: Beacause we cen't spaall

By Gal "Blackbeard" Koplewitz, "Dread Pirate Zac" Bathen, "Peg Leg Meg" Knister, Limor "Land Ahoy!" Gultchin

[Our Fantastic Demo Video]

Overview

Over the past couple of weeks we planned, designed, and implemented an autocorrect application, which has an interactive interface with the user. Each time the user enters a word and presses the space bar, autocorrect checks if that word is spelled correctly, and if not, replaces it with a likely alternative. Below is a summary of how it works.

- A first-time user will have to save a txt file containing text sources he wishes the dictionary to use as "reservoirs" of correct vocabulary.
- When running the program for the first time, we expect the "installation" or construction of the BK-tree (which was our chosen data-structure) to take anywhere from a few seconds to a few minutes, depending on the size of the txt source.
- After this, the program will automatically correct misspelled words that are similar (up to 2 "edit-distances" away) to words found in the installed dictionary, or it will mark the misspelled word with a red tag if the word is not similar enough to any word in the dictionary.

Planning

We planned to have 3 modules: a GUI interface/event handling module, a dictionary module with the implementation of the BK-tree, and a correction module with the algorithms for replacing words. Generally, this proved to be a good plan, and we hit our

milestones when we expected to – in particular, writing out signatures proved to be very helpful in splitting up the work between us. We did, however, modify our plan slightly in that we ended up including the GUI interface/event handler module in the main file, instead of being a separate module.

Design and implementation

Here's a bit about our experience with design, interfaces, languages, systems, testing, etc.

Implementing dictionary module:

This was fairly straightforward. After researching how BK trees work, we found algorithms for how to add elements to the tree, as well as for how to search for an element in the tree. We converted these algorithms into code, and created the "add" and "query" functions. The **add function** takes in a **word and the existing BK-tree** – then adding the word if it is not there, or increments the frequency of the word if it already exists. The **query function** takes in a **word, w, the existing BK tree, t, and the edit distance, d**. It searches for w in t and returns a list of tuples. Each tuple contains a word, its edit distance from w, and the frequency of its use. We originally ran into a bit of trouble with query, because we wanted to return only the tuple of the word, its frequency, and edit distance 0 if the word being queried was already in the tree. However, query was recursive, and the recursive functions were preventing the word from being immediately returned from the tree if it was found. We solved this problem by calling an exception if the word was found, and catching the exception later, allowing the recursive calls to finish before attempting to return anything.

Implementing main/event handler/GUI:

To implement the GUI, we imported a Python addition called Tkinter. Tkinter allows the creation of a message box in which the user can type text. We created an event every time the space bar was pressed, and then **treated the letters between each space bar as a word**. Then we ran functions from our correct and dictionary modules on that word to see if it was in the dictionary and **correct** it if it wasn't. One problem was that the first word

entered was not being processed at all; it was simply left uncorrected if spelled wrong. This is still a problem; we are unsure what led to it, or how to fix it. Another problem was that capitalized words and words with punctuation after them were being treated as misspelled because all words in the dictionary are lowercase and punctuation-free. We solved this problem by converting all words to lower-case and stripping them of punctuation (using common Python functions) before searching for them in the dictionary – and now there should not be any problems.

Implementing correction module:

Query returns a long list of words within edit distance 2 of the misspelled word (we decided this edit distance was the largest reasonable number) that could potentially replace it. To improve the efficiency of the program, we cut down this list of potential replacement words to a list of length at most ten. We wrote a function that created a smaller list out of the 7 words of edit distance 1 with the highest frequencies, and the 3 words of edit distance 2 with the highest frequencies. Then these 10 words through a formula that weighed edit distance and frequency to determine which word was the likeliest replacement for the misspelled one.

The most challenging part of this module was determining the coefficients for these two different factors. We experimented with assigning different values to edit distance and frequency. Eventually, we decided on the function **(frequency)/(edit distance)^4**, which seemed to work relatively well. Initially, we wanted to include keyboard proximity as part of our accuracy algorithm (the idea being that a user is more likely to mistype a letter that is closer to the correct letter on the keyboard than one that is far). However, while keyboard proximity in itself was not too difficult to implement (looking at the keyboard as a 2-dimensional array and calculating "Euclidian" distance between keys is one simple way to do this), we were not sure how to weigh it with regard to the other factors.

Reflection

Improving Accuracy – what did the user *want* to write?

The weighing mentioned above was perhaps the most significant hurdle. For example, if we type "recieve" with the intent of writing "receive," it currently corrects to "relieve". This is due to our considerations of frequency and edit distance being too limited for a comprehensive spellchecker, which turns out to be a non-trivial problem. To truly evaluate the success-rate of our spellchecker, we would have had to construct a huge database of misspelled words with "expected" outcomes, and see what percentage of them came out as true – which was outside the scope of our humble project.

However, we do have estimations for two factors that might improve future spell-checkers:

1) Machine learning from user preferences and habits. We currently have a simple version of this, which increases the frequency of words as the user types them in. However, future versions could automatically add wrods that are used frequently, infer common usages from sentence structures, etc, and in general make use of more advanced NLP tools.
2) Another thing we didn't do was take into account context – the words before and after the misspelled word – as influencing the outcome. This is obviously important in making autocorrect more accurate. Google, for example, have a database of sequences up to 5 words long, which they use in their autocomplete and "did you mean ____?" algorithms.

Other notes

1) We were pleasantly surprised by the ease with which we were able to create events and event handlers. Tkinter is awesome.
2) If we had more time, there are several improvements we could make:
    - Initially, we wanted to make our auto-correct turn words into pirate slang. We would hardcode corrections that turn "Hello" into "Ahoy", "friend" into "matey," and so on.
    - We could include keyboard proximity into our correction algorithm to improve its accuracy.

- We would do more extensive research on the ratios of edit distance, keyboard proximity, and word frequency that result in the most accurate algorithm for determining which word should replace the misspelled one.
- Possibly make the entire program run more quickly by cutting out various inefficiencies.

If we were to redo this project from scratch, we would still program in Python because it's flexible and well-documented, but we might do more research into determining whether BK trees are the most efficient way to search through a dictionary of words.

Individual Roles

Zac: Constructed BK tree/dictionary module

Gal: Constructed main function and parts of correct module.

Limor: Constructed parts of the correct module and the GUI

Meg: Constructed the GUI

Advice for future students

The most important thing we learned is that it's difficult to write an auto-correct spelling program when you're terrible at spelling to begin with. Just kidding. We also learned that it is important to take advantage of the functions that the language already provides for you. When you're about to write a function, you should look to see if there's one already written that you can use instead, if only as reference (or for learning from their mistakes).