Efficient Autocorrect using Levenshtein Automata

Todd Lubin, Peter Bang, Wilson Qin, Kristen Faulkner

Video

Overview

Our group implemented an autocorrect program that provides suggestions to misspelled words based on both Levenshtein edit distances, keyboard distances, and commonness of the word. One Levenshtein edit distance is one insertion, deletion, or replacement. Our program first compiles a list of words that are within some minimal edit distance of the input word and in the dictionary. It then uses keyboard distances and a commonness index (a measurement of how frequent a word appears) to rank the suggestions based on the total accumulative score. Our command-line user interface allows the user to input either a single word, a sentence, or an entire file. We are pleasantly surprised by how accurate these suggestions have been, and are overall very excited about the efficiency and capability of our program.

Planning

In the very beginning, our group planned on implementing a neural network, and was leaning toward building a convolutional network for handwriting recognition. Here is our draft specification, which we wrote when we were planning on implementing the neural network. However, upon doing research, we were unable to find a paper or pseudocode on the topic that we understood and found interesting. Therefore, we decided to pivot and choose a different topic. We thought that implementing the Levenshtein automata for autocorrect would be interesting. We found that we understood the topic more clearly and found motivation to develop the program. Here is our annotated final specification.

Looking back, our final implementation does differ significantly from our original planning. As we went forward with our implementation, we found places where logic was extremely similar and could be factored out into another layer of abstraction. We also found limitations of our code and responded by restructuring our code to make the algorithm more efficient. Overall, the changes we made were not changes that we were forced to make because of insufficient planning in the beginning, but rather positive changes that we decided to make upon gathering new information, gaining a more solid understanding of the algorithm, and critically thinking about how to best improve the implementation of it. It was crucial to the success of our project that we started writing the code early and allowed

ourselves time to first change projects altogether, and then go through a series of restructures before arriving at a final product.

Design and implementation

The scope of this project could be separated into two core components. The first of these components is Levenshtein automata. What was needed from this part of the project was ultimately a function that when given some representation of a dictionary along with a string and an integer, k, would return back a list of words contained in the dictionary within an distance distance of k from the input string. The design of this was a little tricky to get right. We implemented a module Levenshtein that was a functor taking in an Nfa and a Dfa module. The basic Levenshtein algorithm consisted of first building an Nfa representation from a given word and edit distance. The idea was to have an automation that could detect whether a given input word to the automation was within a certain edit distance of another word. This Nfa then needed to be converted into a Dfa using a powerset construction. The Dfa would be much easier to work with in the final step which involved finding strings that satisfied the Dfa and were in the dictionary. The reason why the to_dfa function was put into the Levenshtein module was that it needed to know about both the nfa and dfa abstract types so that it could convert between them.

Another major design decision was in creating a more general Automata module, a functor that took in a bunch of modules defining the states and transition types of the automation along with modules that dealt with creating sets and dictionaries of the states and transitions. With this layer of abstraction, the basic functionality that was inherent in all automations (i.e. ways to build an automation and ways to interact and get information about an automation) could be factored out and only written once. Because the types of our Nfa and Dfa modules were inherently dependent on each other, all the types associated with the two modules were defined in a file called type.ml.

Another design decision was on our representation of the dictionary. The two choices we were contemplating were some kind of trie representation or simply an array representation. Both had their benefits. A trie would allow for constant time lookup for words and could we traversed in tandem with the dfa in order to efficiently find the overlapping words. However, the data structure would take up more more memory and take more time to construct. An array representation would be faster to build up and quicker to implement. It allowed for an easy “zig-zag” method of traversal with the Dfa in order to find matches. This method consisted of being able to find the next valid string that satisfied the Dfa along with being able to find the next string in the dictionary (inclusive). The latter (array) representation was implemented at first because it was quicker to implement, and then we ultimately decided to stick with the representation because the algorithm was already as efficient as we had hoped and the costs of the immense memory consumption of a trie representation of a dictionary outweighed the benefits of added efficiency.

The second major aspect of the project dealt with processing the results from the Levenshtein algorithm and somehow ranking them in an appropriate and logical way. The obvious way to output matches is to give the words with the smallest edit distance away the highest priority. This has its obvious flaws as it

doesn't take into consideration many factors that we found to be crucial in giving accurate suggestions. The first of these suggestions was word popularity. For words with the same edit distance away, more popular words should clearly be ranked as a better suggestion. What about if the more popular word was just one more edit away? The ranking algorithm would also have to take into consideration how bad the edits were. If there was a letter swap of letters that were across the keyboard, it is not likely that this was a typo. However, with two letters next to each other on the keyboard, it is much more common to have a character swap. Ultimately, these two factors of popularity and keyboard similarity would have to be combined in some way to create a "score" that could be used to sort the results.

The design of the scoring algorithm utilizes a module to deal with similarity based on keyboard distances, in qwerty.ml, a module to deal with word popularity, in ranker.ml, and a module that combines the two, in score.ml. The design of this was intentional so that other parts of the project only need to know and interact with the Score module and not the modules that the Score module relies on to compute its scores. The Ranker module is used to associate a given word with a frequency. A file of 50,000 words along with a frequency index is loaded into a dictionary. This frequency index is not perfect but gives a good sense of words that are more popular in the language than others.

One of the more interesting and unique aspects of the whole project comes in how keyboard similarity is determined. In the QwertyKey module, the following heuristic was created to determine keyboard distance: first break up the two words that are being compared into two halves. The first halves of both words are compared and get some distance value. Then the second halves are compared to generate another distance value. These two distances are added to determine the final distance between the two strings. To determine the distance between two strings, find the maximum overlapping sequence. For example, the strings "tser" and "sero" have a maximum overlapping sequence where the "t" is overhanging left, the "ser" are aligned and the "o" is overhanging right. The distance is then computed by penalizing overhangs and determining the keyboard distance between the aligned characters. Although this is a quadratic operation, it was thought to be acceptable as the strings were split in half and wouldn't be called on large values of n. This heuristic was created in order to be able to handle insertions and deletions in the beginning, middle, and end of words so that matching patterns in two strings could be aligned and not incorrectly penalized. The match between two strings was then a scaled version of the total distance so that values were always between 0 and 1. The combination of word popularity and our string keyboard heuristic that we developed gave two valuable metrics that could be combined to arrive at a total ranking score.

Efficiency Analysis

		String length						
		1 ("a")	2 ("be")	3 ("the")	4 ("back")	5 ("place")	6 ("market")	7 ("brisket")
Edit Distance	1	77 (98.72%)	93 (73.23%)	128 (73.56%)	158 (72.15%)	139 (53.05%)	197 (65.02%)	130 (38.01%)
		78	127	174	219	262	303	342
	2	1496 (56.37%)	1974 (44.31%)	2187 (35.78%)	3277 (42.94%	2585 (28.66%)	3802 (36.99%)	2667 (23.35%)
		2654	4455	6112	7631	9018	10279	11420
	3	12609 (14.90%)	15255 (10.63%)	18494 (9.46%)	23623 (9.80%	21968 (7.82%)	26284 (8.34%)	21421 (6.21%)
		84630	143479	195414	240999	280774	315255	344934

In implementing a Levenshtein automata, we not only are able to efficiently find a given set of words that are within a certain edit distance of an input word, but we also have an efficient way of probing a dictionary to determine which of these words are in the dictionary. Using a brute force approach, a "dumb" algorithm may involve getting a list of words within a certain edit distance and checking every one of these words in the dictionary. Levenshtein allows us to probe the dictionary only a fraction of this many times. The previous data table shows how the fraction of the number of probes in the dictionary the Levenshtein automata must perform changes as a function of string length and edit distance. As you can see, a noticeable improvement is already seen with strings as short as "the" and an edit distance of 2. It is not unreasonable to have to go out to edit distances of 3 or 4, in which case Levenshtein only has to probe the dictionary in less than 10% of the words the "dumb" algorithm would have to for short strings. The time savings increase dramatically with longer words and further edit distances. Also, the algorithm can be made even more efficient by using a smaller dictionary. These tests were run on a dictionary of size of more than 200,000 words.

Reflection

In reflection, the project came out very well. We not only implemented the Levenshtein automata data structure and algorithm in order to determine words within a certain edit distance of an input string, but we also applied it to a real world problem: auto correction. In doing so, we implemented a ranking algorithm to give accurate and smart suggestions based on word popularity as well as keyboard layout. The heuristics that we developed were well thought out and gave surprisingly good suggestions in our testing. The final product ultimately works well, and is polished so as to be easily applied to input words, sentences and files. The code is written efficiently and structured clearly so that it can easily evolve if this project is continued.

Getting to this final product took a lot of systematic planning and thought. The code was divided into logical sections with clear interfaces and modules. This was crucial in the testing of the code. Each module was tested independently of the others and then brought together. One of the main ways that the automation modules were tested was with a handy function that printed an automata in an easy to read format. Using this function, it was very easy to build up an automation and print it to make sure all the methods used were bug free. Another great way that the code was kept bug free was by using many layers of abstraction. With well thought out type definitions, many functions became almost trivial in nature. For example, the entirety of the automata.ml file needed very little testing. All the functions to build and

interact with an automata simply involves adding or checking whether certain states or transitions were in different sets and states.

Many times we were faced with the question of how much abstraction to use. In one way, abstraction is great in making code more readable and easier to maintain. In another sense, sometimes using too much abstraction can result in inefficient functions. It took a lot of deep thought and a couple structural changes to come up with what we think is a good balance between the two.

If we had more time, we would try to cover up some of the pitfalls of using Levenshtein automata for spell checking. There are good reasons why spell checking algorithms for Google and Microsoft, for example, make use of a lot of machine learning and large amounts of data concerning commonly misspelled words. Levenshtein automata is often criticized for its inability to deal efficiently with words that include swapped characters - for example, “ptifall”. This would count as two edits when intuitively it should count as just one. Ideally, we could somehow combine machine learning with Levenshtein automata to create an autocorrect tool that could perform well on all types of misspellings. It also would be interesting to be able to evolve the frequency index so that each individual user could have different rankings based on their individual language patterns.

Advice for future students

Planning ahead is very important. Giving yourself unreasonable deadlines and checkpoints to have certain functionalities can motivate you to work hard to keep yourself on track, and even if you don’t meet these deadlines exactly, you will still be successful overall. It is important to think about the abstraction and the interaction of your code from very early on. That being said, you should not be afraid to change the structure of your code. As you learn more about your implementation and run some tests, you will realize that your initial draft was not perfect - and it is perfectly fine to change directions!

Also, making clear interfaces is extremely important, especially in a group project. Interfaces will make it clear what additional functionality needs to be implemented and how the work can be efficiently divided among the group members. In addition, a lot of time and code can be saved by first writing code at a high level. This means writing code in terms of functions that haven’t been written yet. When this is done, it is very clear what functions need to be written for the whole project to come together.