Fuzzy String Matching Applied to Spell Correction

Work by Casey Kane and Jonathan Terner

Introduction and Background:

Imagine that you are at a friend's house helping him make Christmas cookies. You've never been to his house before, so you are going to need a little direction if you are going to be of any help. Your friend asks you for the red cookie cutter that is shaped like a star, so you go into the cabinet to find it. You search the entire cabinet, however, and the only cookie cutter that is star shaped is blue--not red. What do you do? Well, clearly your friend wants to make star shaped cookies, so you might as well bring him something, even if it's the wrong color.

This is a prime example of fuzzy matching. Given a certain description of an object, we want to see if we can find it amongst some collection of candidates. If you can't find the object of that exact description, we want to find something that is *close*. In the above example, you give your friend the closest item to his description that you can find; you've assumed that the item in the cabinet with the closest shape was the best match.

In this project, we implement fuzzy string matching and explore its application in spell checkers. Computers are very good at comparisons, and it is easy to test the equality of two strings on a character by character basis. This, however, is very limiting. Rather than compute whether two strings are different, you might also want to compute the similarity between them. To compute the similarity between two strings, we can use a metric known as Levenshtein distance. Levenshtein distance (or "edit distance") can be defined as the the minimum number of edits needed to transform one string into another, where an edit is a character deletion, replacement, or insertion. In our project, we say that **the smaller the edit distance, the more similar the words**.

Problem Statement:

Given a word, determine if it is spelled correctly. If not, offer the "best"correction that is within two edit distances of the original word. If there are no words within two edits, then we say that we can't correct the word.

Algorithm Description:

Overview:

There are two main goals that our algorithm tries to meet:

- Speed
- Quality of correction.

To implement a spellchecker, the following are needed:

- Language model
- Error model

A Language model is essentially a sample of a written language, which in this case is English. We build our model by reading and processing the word frequency from the books Pride and Prejudice, War and Peace, and The Adventures of Huckleberry Finn (4.2 MB).

The Error model is used to understand and differentiate spelling mistakes. The error model is used to maximize the following probability:

$$P(correction \mid misspelled\ word)$$

In other terms, the correction that we provide is the word that is the most probable candidate given a particular misspelled word.
We make the following assumptions:

- Likelihood of a correction candidate is determined by its count
- Words within 1 edit distance are assumed to be 100 times more likely than words that are two edit distances away and are weighted accordingly.

Edit distance implementation:

To compute edit distance between two words, we use a dynamic programming approach. This problem has an optimal substructure. Given two strings A and B of length an X and Y respectively, the edit distance between A and B can be derived by computing the edit distances of all their substrings that can be obtained by removing characters off of their ends.
This is the iteration relation:

```
editDist(x,y) =:
    -   max(x,y) if min(x,y) = 0 //inserting all characters of one string into the other
    -   otherwise min of:
            -   // min of deleting given character from A
                editDist(x-1, y)
            -   // min of deleting given character from B
                editDist(x, y-1)
            -   // min of deleting both  characters (replacement)
                editDist(x-1,y-1) + (1 if A_x != A_y, 0 if A_x = A_y)
```

The dynamic approach is particularly useful because it is not necessary to compute minimum number of all possible permutations of edit operations, which would be very expensive. This algorithm can also be considered as a distant cousin of longest common subsequence, with the key difference being that longest common subsequence does not allow substitution. The time complexity for this step is always $\Theta(|A||B|)$.

It is interesting to note there exists bounds on the edit distance between two words:

- The edit distance is at most the length of the longer string
- The edit distance is at least the difference in lengths of the two strings.

We make use of these bounds in the Correction lookup implementation.

Correction lookup implementation:

As explained above, our language and error models are based upon samples of text. For our purposes, the language model is used for the following:

- Look up if a given word is spelled correctly
- If the word is not in the model, find a list of words within a certain edit distance

The language model is obtained by reading the sample text in from a file, obtaining a unique list of words, and then sorting these words in a way that will allow for quick access and iteration times. The data structure used for the language model is a vector of unordered maps. The unordered maps at each index contain words of the a certain length (i.e. index 0 contains words of length 1, index 1 contains words of length 2, and so on). The data structure allows for constant access time $\Theta(1)$ in the average case when searching for an existing word. Additionally, since the model is sorted by length and iteration over unordered maps is quite simple, iteration over our data structure can be done in $\Theta(N)$ time in the worst case, where N is the number of words in the language model. For our purposes, the error model is used for the following:

- Determine the probability of a word occurring in the sample text

The error model is similar to the language model in that it holds a unique set of words from a sample text. The difference is that when a word occurs more than once, the error model increments the count of the word. The result for each word is the total occurrences of the word in the sample text stored alongside the word. (<u>NOTE</u>: Large initial cost to create dictionaries, however only happens once)

Now that the models used have been defined, we can explain in detail the algorithm we used to determine the correction for a given word:

(1) Given a word, determine if it exists in the language model (this step is completed in $\Theta(1)$ time)
(2) If the word exists in the model, proceed to step 5.
(3) If the word does not exist, find all words that may be found within a 2 edit distance of the given word. In this step we simply iterate over all words in the language model that are between length-2 and length+2, where length is simply the length of the given word. We

check each word's edit distance with respect to the given word, and if it is 2 or less we add it to the candidates (NOTE: we assume computing the levenshtein distance is a constant time complexity since the longest word in the english dictionary is 45 characters). If a correction candidate violates this length bound, then it violates the bounds of Levenshtein distance and cannot be found within 2 edits. The iteration is at worst $\Theta(N)$ but only occurs when all words in the model are within range (NOTE: If this occurs, it is highly likely the language model used is bad). When this step is done, we should have a list of words within 2 edit distance of the word we need to correct.

(4) Now we iterate through the list to determine the word with the highest probability in the error model. We determine this by the total proportion of the word within the error model. (IMPORTANT NOTE: Words with a smaller edit distance are given a higher weight in order to increase the likelihood the algorithm will give a better correction). For this analysis, we will consider the calculation of edit distance to be constant with respect to the size of the Dictionary.

(5) Return the correction to the user

The total time the algorithm will run is $O(N)$ in the worst case, where N is the number of words in the language model.

Results and discussion:

Total Complexity:

As stated above, since the length of the longest word in the English dictionary is finite, the worst case asymptotic growth of our algorithm with respect to an arbitrary dictionary size is $O(N)$.

Results of testing:

We observed a high start up cost when building the language and error model (about 1 second). However, once the main user loop starts, our program was very fast in offering corrections. Some words that are indeed correct English words do not correct, as they were not found in our particular language model.

Further Work and Improvements:

As a spellchecker, our solution is simple, as we ignore many of the complexities of written language. Our assumption that words that are within one edit distance are more probable corrections than words that have greater edit distances is not always true. A better error model would consider word context. Word context is the idea that certain words have different probabilities depending on the words that surround them. For example, in a certain context, some corrections might cause a grammatical error in a sentence, whereas other corrections might not. Our solution was aimed at showing applications of fuzzy string matching, and was not meant to be a robust spell checker.

References

This work is roughly based on the work done by Peter Norvig at http://norvig.com/spell-correct.html. Our implementation is different as we try to find a more efficient solution than the one Norvig presents. Our idea of using edit distance is roughly the same as Norvig's, except that we make use of the computation of edit distance whereas Norvig only uses the concept without the calculation.