**SPELL CHECKER**

Submitted by:

Sumit singh 11713099

Ravi kumar 11717681

Acknowledgement

I would like to express my special thanks of gratitude to my teacher Ishan Kumar who gave me the golden opportunity to do this wonderful project on the topic spell checker, which also helped me in doing a lot of Research and i came to know about so many new things I am really thankful to them.Secondly i would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Abstract

This project addresses on the problem occurs in modern spell checking: realword error detection and suggestion. This project has taken a classic word trigram approach and a mixed part of speech trigram approach as real word

error detection method. The suggestion process is done by confusion sets constructed by phonemes, word distances, word permutation and some self defined confusion sets.

Introduction

**Definition**

What is spell checking? Date back to 1980s, a spell checker is more like a “verifier”[1]. It has no corresponding suggestions to the spelling error detected. As many of the readers are using word processor nowadays, a spell checker will first mark a word as mistaken(Detection) and give a list of replacement of word(Suggestion). Therefore the definition of spell checking involve more than only *checking* , it is the process of *detecting*

misspelled words in a document/sentence and *suggest* with a suitable word in the context.

Therefore, to construct a spell checker, it needs to have the following features:

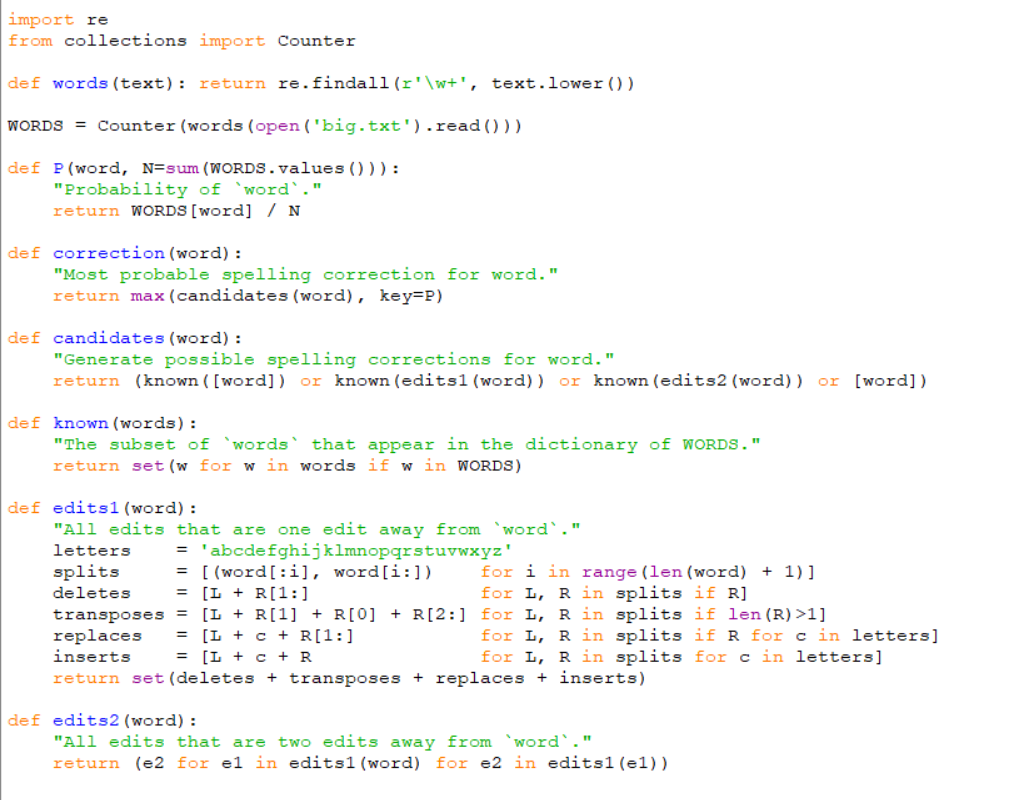
1. Spelling Detection: the ability to detect a word error

2. Spelling Suggestion(Correction): the ability to suggest a suitable word to users which matches their need in context

**A basic spell checker: Dictionary check**

In many classic approach, a spell checker implements a simple dictionary check structure.The overall implementation involves simple dictionary check. A diagram demonstrating this algorithm is showed in Figure 1.

This method is nice and easy, also requires a low level of programming. Developer can simply define a sets of dictionary words for spelling detection and suggestion. If an spelling error occurs, do binary search on the dictionary list and generates a number of corrections.To improve accuracy, simply expanding the dictionary size and it can detect more words.



**How It Works: Some Probability Theory**

The call correction(w) tries to choose the most likely spelling correction for w. There is no way to know for sure (for example, should "lates" be corrected to "late" or "latest" or "lattes" or ...?), which suggests we use probabilities. We are trying to find the correction *c*, out of all possible candidate corrections, that maximizes the probability that *c* is the intended correction, given the original word *w*:

argmax*c ∈ candidates* P(*c*|*w*)

By [Bayes' Theorem](http://en.wikipedia.org/wiki/Bayes'_theorem) this is equivalent to:

argmax*c ∈ candidates* P(*c*) P(*w*|*c*) / P(*w*)

Since P(*w*) is the same for every possible candidate *c*, we can factor it out, giving:

argmax*c ∈ candidates* P(*c*) P(*w*|*c*)

The four parts of this expression are:

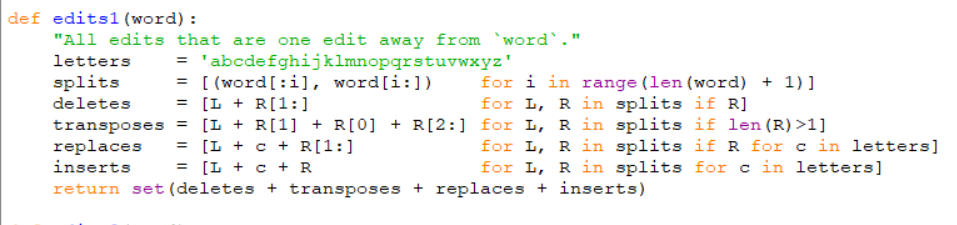
1. **Selection Mechanism**: argmax   
   We choose the candidate with the highest combined probability.
2. **Candidate Model**: *c ∈ candidates*  
   This tells us which candidate corrections, *c*, to consider.
3. **Language Model**: P(*c*)   
   The probability that *c* appears as a word of English text. For example, occurrences of "the" make up about 7% of English text, so we should have P(*the*) = 0.07.
4. **Error Model**: P(*w*|*c*)  
   The probability that *w* would be typed in a text when the author meant *c*. For example, P(*teh*|*the*) is relatively high, but P(*theeexyz*|*the*) would be very low.

One obvious question is: why take a simple expression like P(*c*|*w*) and replace it with a more complex expression involving two models rather than one? The answer is that P(*c*|*w*) is *already* conflating two factors, and it is easier to separate the two out and deal with them explicitly. Consider the misspelled word *w*="thew" and the two candidate corrections *c*="the" and *c*="thaw". Which has a higher P(*c*|*w*)? Well, "thaw" seems good because the only change is "a" to "e", which is a small change. On the other hand, "the" seems good because "the" is a very common word, and while adding a "w" seems like a larger, less probable change, perhaps the typist's finger slipped off the "e". The point is that to estimate P(*c*|*w*) we have to consider both the probability of *c* and the probability of the change from *c* to *w* anyway, so it is cleaner to formally separate the two factors.

**How It Works: Some Python**

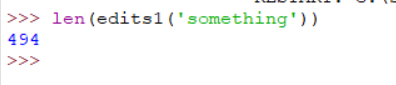
The four parts of the program are:

1. **Selection Mechanism**: In Python, max with a key argument does 'argmax'.
2. **Candidate Model**: First a new concept: a **simple edit** to a word is a deletion (remove one letter), a transposition (swap two adjacent letters), a replacement (change one letter to another) or an insertion (add a letter). The function edits1 returns a set of all the edited strings (whether words or not) that can be made with one simple edit:

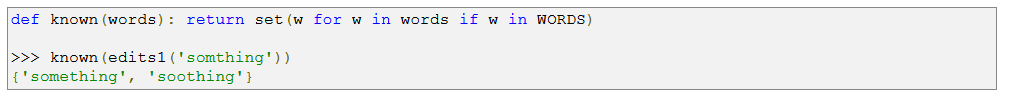


This can be a big set. For a word of length *n*, there will be *n* deletions, *n*-1 transpositions, 26*n* alterations, and 26(*n*+1) insertions, for a total of 54*n*+25 (of which a few are typically duplicates). For

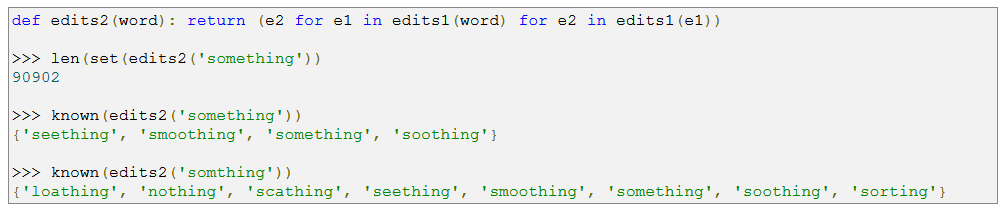
Example ,



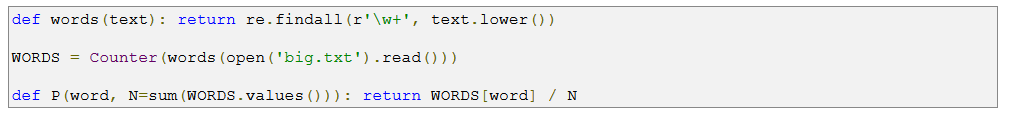
However, if we restrict ourselves to words that are *known*—that is, in the dictionary— then the set is much smaller:



We'll also consider corrections that require *two* simple edits. This generates a much bigger set of possibilities, but usually only a few of them are known words:



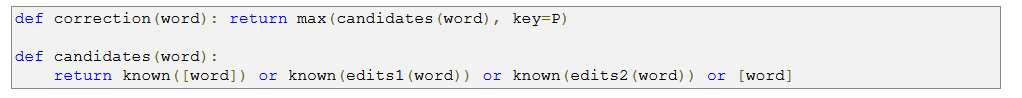
We say that the results of edits2(w) have an **edit distance** of 2 from w.

**3. Language Model**: We can estimate the probability of a word, P(word), by counting the number of times each word appears in a text file of about a million words, [big.txt](https://norvig.com/big.txt). It is a concatenation of public domain book excerpts from [Project Gutenberg](http://www.gutenberg.org/wiki/Main_Page) and lists of most frequent words from [Wiktionary](http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists) and the [British National Corpus](http://www.kilgarriff.co.uk/bnc-readme.html). The function words breaks text into words, then the variable WORDS holds a Counter of how often each word appears, and P estimates the probability of each word, based on this Counter: ****We can see that there are 32,192 distinct words, which together appear 1,115,504 times, with 'the' being the most common word, appearing 79,808 times (or a probability of about 7%) and other words being less probable:



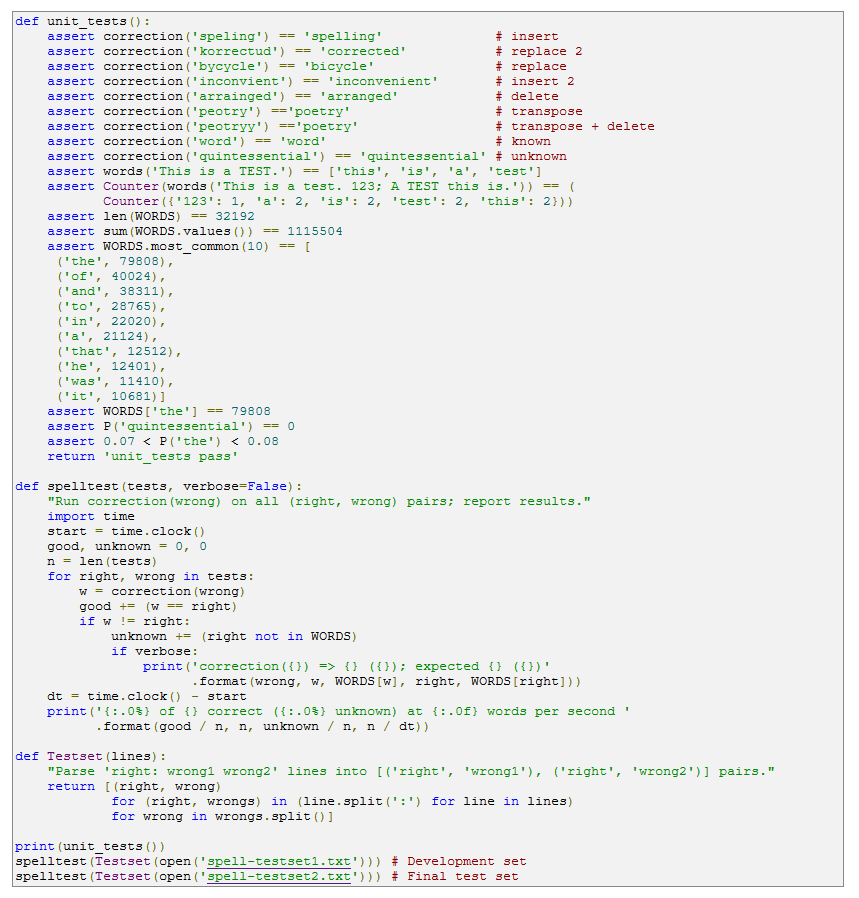
1. **Error Model**: When I started to write this program, sitting on a plane in 2007, I had no data on spelling errors, and no internet connection (I know that may be hard to imagine today). Without data I couldn't build a good spelling error model, so I took a shortcut: I defined a trivial, flawed error model that says all known words of edit distance 1 are infinitely more probable than known words of edit distance 2, and infinitely less probable than a known word of edit distance 0. So we can make candidates(word) produce the first non-empty list of candidates in order of priority:
2. The original word, if it is known; otherwise
3. The list of known words at edit distance one away, if there are any; otherwise
4. The list of known words at edit distance two away, if there are any; otherwise
5. The original word, even though it is not known.

Then we don't need to multiply by a P(*w*|*c*) factor, because every candidate at the chosen priority will have the same probability (according to our flawed model). That gives us:



## Evaluation

Now it is time to evaluate how well this program does. From that I extracted two test sets of corrections. The first is for development, meaning I get to look at it while I'm developing the program. The second is a final test set, meaning I'm not allowed to look at it, nor change my program after evaluating on it. This practice of having two sets is good hygiene; it keeps me from fooling myself into thinking I'm doing better than I am by tuning the program to one specific set of tests. I also wrote some unit tests:



This gives the output:



So on the development set we get 75% correct (processing words at a rate of 41 words/second), and on the final test set we get 68% correct (at 35 words/second). In conclusion, I met my goals for brevity, development time, and runtime speed, but not for accuracy. Perhaps my test set was extra tough, or perhaps my simple model is just not good enough to get to 80% or 90% accuracy.