# Web Information Retrieval:
# Spell Correction Implementation

Shahaf Hermann

Video Presentation

Final Project in the course: Web Information Retrieval (67782)

School of Engineering and Computer Science
The Hebrew University of Jerusalem

Teacher: Prof. Sara Cohen

July 2020

## ABSTRACT

This paper describes the final project in the course "Web Information Retrieval", which focuses on spell correction. The implementation of this project takes into account the differences between typographic errors and phonetic errors and operates under the assumption that the typist knows what words he or she wants to type, but some noise is added in the form of typos and spelling errors. Using bigram comparisons, the program eliminates candidates that can't possibly be a match for the intended word. Next, the new candidates are filtered using Jaccard Coefficient, and a Damerau-Levenshtein distance. Finally, the program branches to two optional endings, which attempt to provide solutions for the two error types – The first one applies a noisy channel with Bayes rule, using confusion matrices and the probabilities of writing a specific letter or having a specific spelling error. The second one searches the history of previous queries and uses this information to decide on the best correction. Experimentation with these algorithms showed better running times for the history search (on average).


## INTRODUCTION

When first approaching this project, I was looking for a subject that I would find both interesting and useful. Spell correction is an excellent example for a field of study that is integrated into our everyday lives - whether we know it or not - and is a fundamental component of almost every program that contains a text editor, such as Microsoft Word, Google's search box or even the SMS application's text field on our smartphone.

The first problem we should acknowledge about spell correction is that it could be interpreted in multiple ways. This means that a single spelling error could have many possible corrections, all of them are valid grammatically. For example, the misspelled word "drenk" could be corrected to "drink" or "drunk", and both would be correct in one situation or another – so the context is important.

The second problem we should keep in mind is that spelling errors could happen for multiple reasons as well. Spelling errors are usually divided into two broad categories[1]: (1) typographic errors, which occur when accidently typing a wrong letter, adding letters to or deleting letters from the word we intended to

type, and (2) phonetic errors, where the misspelled word is pronounced the same, but the spelling is wrong. Phonetic errors are harder to correct because they distort a word with more than one insertion, deletion or substitution. In this case, we would like to perform a search for close matches, that would sound the same.

This project attempts to solve the second problem, by implementing a Noisy Channel with Bayes Rule algorithm and a Search History algorithm to provide corrections for typographic errors and phonetic errors, respectively.

# METHODS

## General

For the implementation of this project, I have added to the previously done project 776 new lines of code, span over 5 Java classes and 44 methods. The majority of the main spelling correction algorithm (figure 1) is following the same steps of pre-processing for both the typographic and phonetic errors, and only branching towards the end in order to apply the different logic that handles those different error types.

## N-gram Language Models

An n-gram is a contiguous sequence of n elements from a given sample of text or speech. For this project's purpose, an n-gram is a sequence of n letters of a word. This model becomes very useful when trying to focus the search to words similar to the intended misspelled word. When comparing the n-grams of two words and filtering them using Jaccard Coefficient (The intersection of sets of n-grams of each word, divided by the union of those sets), we are left with a very small list of potential correction candidates, which we could then perform more processing on.

In order to do so, I had to create an index of all available n-grams in the corpus and remember to which words they belong. Thanks to the fact that the alphabet is finite (and rather small), and because the model used in this project is a bigram (2-gram), having all bigrams of the corpus calculated wasn't hard – Let N be the size of the alphabet. Then there are $\binom{N}{2}$ different options for bigrams in the corpus. Let's assume that N=30 (English alphabet size, plus a few special characters), then $\binom{30}{2} = 435$ which isn't much. Furthermore, this

process only happens once on index creation, so it doesn't affect the search speed. The downside to this method is that it consumes a non-neglectable amount of disk space.

```
findSpellingCorrection(term, historySearch):
    ngrams <- All n-grams of term
    common <- All terms in the corpus that have at least
                  one n-gram in common with term
    jaccard <- Out of the common terms, get the terms
                   with Jaccard Coef. higher than Threshold
    dld <- Out of the Jaccard terms, get the terms with
                DL distance lower than a fixed minimum.

    if historySearch is True:
        best <- Find if any of the DLD terms were Queried
                before, and use the most common one.
    Else:
        best <- Apply the Noisy Channel with Bayes Rule on the
                DL distance terms.

    return best
```

**Figure 1:** Main spelling correction algorithm pseudo code.

## Damerau-Levenshtein Edit Distance

The edit distance (or Levenshtein distance) between strings $a_1 \cdots a_m$ and $b_1 \cdots b_n$ is the minimum number of atomic operations (insertion, deletion and substitution) that are needed to transform one string into the other. This algorithm is known to run in time complexity of $O(mn)$ [2, 3].

Damerau-Levenshtein distance adds an additional operation, which is the transposition of adjacent characters [3]. This operation allows to reduce the distance between certain words, making them more viable for correction.

In addition, DL distance is a dynamic programming problem, so it's algorithm can be "reversed", in order to find the correct set of moves that lead to the solution. This fact is very important, and came in handy when I implemented the Noisy Channel algorithm since it requires knowledge of the type of edits made to the misspelled word in order to transform it into the suggested correction.

This part of the implementation turned out to be somewhat difficult, since it required not only to calculate the minimum DL distance, but also to remember for each word what edits should be made in order to use it as a

correction. Ultimately, this problem was solved by using the Adapter design pattern – after calculating the DL distance, the method returns a new wrapper object that encapsulates the strings involved in the calculation, as well as the distance matrix that was computed in the process. This way provided a relatively easy and fast access to both the minimum distance and the list of edits required (implemented as a method of this class). Each edit should provide information of its type (e.g. substitution) as well as the index where it happened in the suggested correction string, so in order to allow this kind of functionality each edit was represented as a new Edit object.

## Noisy Channel with Bayes Rule

The noisy channel model is a matrix $\Gamma_{xw} = \Pr(x|w)$, where $w$ is the intended word (the correction), and $x$ is the misspelled word. This probability is calculated as an independent product of the probabilities of having each of the edits required for converting $x$ into $w$. Each of these conditional probabilities is calculated from one out of four confusion matrices [4]: (1) $del[x, y]$, the number of times the characters $xy$ (in the correct word) were typed as $x$; (2) $ins[x, y]$, the number of times that $x$ was typed as $xy$; (3) $sub[x, y]$, the number of times that $y$ was typed as $x$; (4) $trans[x, y]$, the number of times that $xy$ was typed as $yx$. The probabilities are then estimated by dividing by the corresponding value of $count[x]$, the number of times that $x$ appeared in the corpus. let $p_i$ be the probability for having some edit, Then:

$$p_i = \begin{cases} \dfrac{del[w_{i-1}, w_i]}{count[w_{i-1}, w_i]}, & if\ deletion \\ \dfrac{ins[w_{i-1}, x_i]}{count[w_{i-1}]}, & if\ insertion \\ \dfrac{sub[x_i, w_i]}{count[w_i]}, & if\ substitution \\ \dfrac{trans[w_i, w_{i+1}]}{count[w_i, w_{i+1}]}, & if\ transposition \end{cases}$$

Finally, the values assigned to $p_i$ are smoothed using the Add-1 smoothing method.

In order to finish the calculation, for each candidate $w$ the value $\Pr(x|w)$ is multiplied by $\Pr(w)$, which is calculated by using a unigram language model. After computing the probabilities for all candidates, pick the one with the highest probability.

Using the confusion matrices and counting letter occurrences throughout the code imposed a minor threat to the efficiency of the program, which I solved by having a single instanced object responsible for all term related probabilities. The Kernighan confusion matrices are well known and thought to be very precise, thus I decided to hard - code them into the *LetterProbability* object. In addition to that, I implemented a method for counting the occurrences of each letter or pairs of letters in the entire corpus. This happens only once when writing the index, and while it does consume a little bit of disk space, it's a very small and fixed amount which almost doesn't change with increased corpus sizes.

```
historySearch(dldTerms):
    history_map <- Load the queries history and count
    best <- ""
    max_count <- 0

    for each term in dldTerms:
        cur_count <- history.get(term)
        if cur_count > max_count:
            max_count <- curCount
            best = term

    return best
```

**Figure 2:** History Search algorithm pseudo code.

## History Search

While the noisy channel with Bayes rule deals very well with typographic errors, it is not the case for phonetic errors. As discussed earlier, phonetic errors are, by nature, more difficult to correct. One way to solve this problem would consist of using a bigram language model and calculating the probability of the correction candidates based on their neighbors in the typed query. Another solution, which I have decided to implement in my project, is keeping record of previous queries, and of the corrections made for them. My estimation was that this way, when encountering a word that is similar to a word that was already queried before, it would be more efficient to look for the candidate in the history and pull the correction from there (Figure 2).

The queries history is saved to a dedicated serializable object, which is saved to a file after each query has finished processing. This history file is loaded when starting the program, along with the rest of the saved index, and deleted when the index is removed.

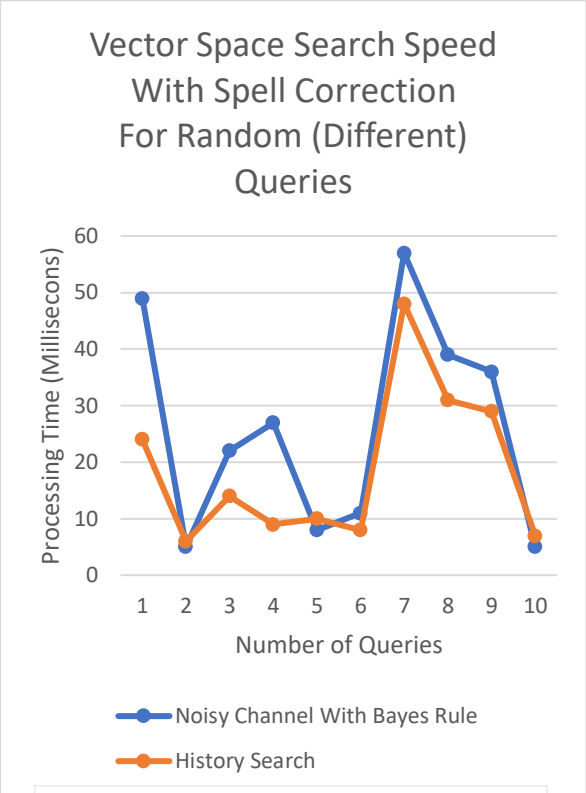
## EXPERIMENTATION & RESULTS

For testing purposes, I have created two lists of queries, each list of size 10. The first list contained queries of random length and content, and the second list contained queries that are similar, in such a manner that each query has at least one common term with the previous query, or a misspelling of that term. The running time was measured for the two search methods that we implemented before (i.e. Vector Space Search and Language Model Search) and does not include index build times (Figures 3-6).

The results for these tests were surprising: when considering random queries, the running time for query processing with History Search was significantly faster than the running time for query processing with a noisy channel with Bayes rule. This result was not anticipated since every new query requires iterating over the whole history. This result could be affected by the small sample size, so as the number of queries grows so is the history size and hence the search time, whereas the running time for the noisy channel would not be affected by the increased number of queries because it's independent of the previous ones. On the other hand, when considering similar queries, the results were inconclusive – I expected the History Search to be much faster since the corrections were already present in the history data base, but apparently it only made the search slower. This could be due to inefficiency of the History Search algorithm or data structure used for holding it.

It is worth mentioning that History Search sometimes provided a better correction for misspelled words. Since the program regards every word that exists in the corpus as a "correct" word, and every other word as a misspelled word, and since the corpus is comprised of user reviews, it might contain words that don't necessarily exist. For example, the word "lo" can be found in the corpus. Consider the following queries: "treats that dogs love", "treats that dogs lo". In the second query, the word "lo" is obviously an error, but it wouldn't be corrected by the noisy channel algorithm. On the other hand, History Search

would find out that the word “love” was used in the past and will use it as a correction for “lo”.




**Figures 3:** Vector space search processing speeds for random queries

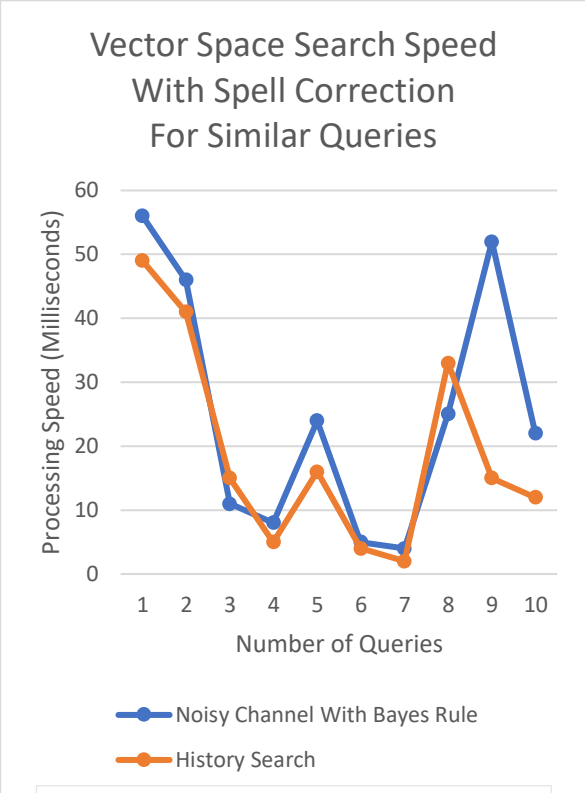



**Figure 5:** Vector space search processing speeds for similar queries

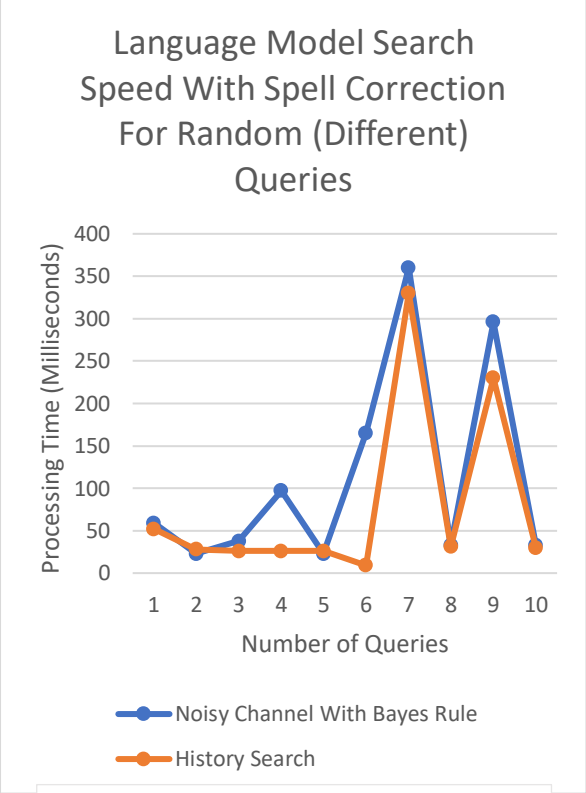



**Figure 4:** Language model search processing speeds for random queries

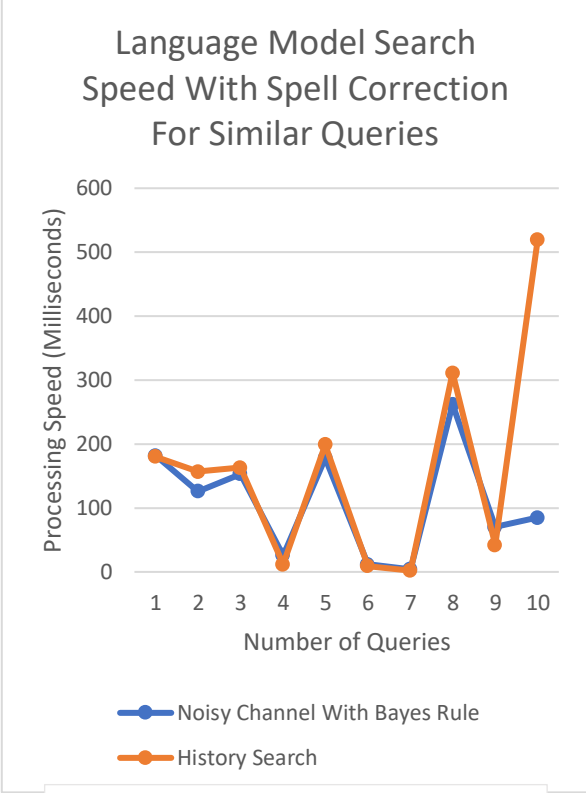



**Figure 6:** Language model search processing speeds for similar queries

# CONCLUSIONS

Spelling correction is a very large field of study, with many branching methods and theorems. As technology advances, this field grows even larger and covers more aspects of our everyday lives.

There are many tools that provide a very strong spelling correction functionality, by using the very same methods implemented in this project. My implementation provides a small glimpse into a very wide and interesting field of study, and although many improvements could be made, it still manages to capture the essence of the subject and provide a functioning error detection and correction program.

As it appears from the experimentation graphs, a good algorithm would be some combination of the noisy channel and the history search algorithms. Future work to improve this project, could include support for n-gram letter search (rather than the current bigram), extending the probability calculations to support bigram language model for learning the context of the word in the query, as well as improving the History Search algorithm by using a more efficient data structure, with regard to both space and time complexities.

## REFERENCES


1. Bruno Martins, Mário J. Silva:
   Spelling Correction for Search Engine Queries. EsTAL 2004: 372-383
2. Esko Ukkonen:
   Algorithms for Approximate String Matching.
   Information and Control, Vol. 64, Nos. 1-3, 1985.
3. Axel Samuelsson:
   Weighting Edit Distance to Improve Spelling Correction in Music Entity Search.
   Masters Thesis in Computer Science, KTH Royal Institute of Technology, 2017.
4. Mark D. Kernighan, Kenneth W. Church, William A. Gale:
   A Spelling Correction Program Based on a Noisy Channel Model.