# UCSD Research Exam (Summer 2004) "Speling Korecksion: A Survey of Techniques from Past to Present" (Final Draft).

Dustin Boswell
*dboswell [at] cs ucsd edu*

February 17, 2005

**Abstract**

This report is a thorough investigation of spelling correction. Papers from 1962 to present are surveyed and an organized tour of spelling correction system internals is given. Lastly, a recent and promising view of spelling correction as an instance of the "noisy channel model" is presented.

## Contents

## 1 Introduction

In writing text, it is desirable to produce a document that is free from grammatical and spelling errors, perhaps as a way to reflect the intelligence of the author or the care with which the document was written. In other cases such as information retrieval, where computers store and retrieve text exactly, it is mandatory that humans make error-free queries. Indeed, one of the earliest papers on spelling correction was in 1962: "Retrieval of Misspelled Names in an Airlines Passenger Record System." ([Dav62]).

Since then, spelling correction systems have become more pervasive, typically a feature of modern text editors. The aim of this paper is to give a survey of spelling correction literature and techniques from the 1960's to present.

First, let us define the **goal of spelling correction**:

> *To assist humans in the detection of mistaken tokens of text, and their replacement with corrected tokens.*

This definition is meant to be loose enough to cover simple systems that just identify mistakes (leaving the task of replacement to the user), to advanced systems that automatically detect and correct mistakes. The ultimate system would make corrections automatically with near perfection as a skilled human editor might.

The term *spelling* correction is technically too specific. In this paper we consider techniques that can be used in the presence of many different errors including typing errors or even non-human errors such as those produced by an optical character recognition (OCR) system. We are concerned with the general problem where a sequence of tokens has been transformed by some error process, and the task at hand is to detect and restore the transformed tokens to their original form.

The paper is organized as follows. In sections 2 & 3, we describe the nature of mistakes in text. Section 4 outlines the typical spelling correction system. Methods for the first step - recognizing mistakes - are described in section 5. Methods for the second step - generating suggestions - are given in section 6. In section 7, we itemize the main difficulties that spell correctors face, and in section 8 we identify improvements and features needed in spelling correction. Section 9 presents the "noisy channel" view of spelling correction, and section 10 concludes.

## 2 Sources of Spelling Mistakes

There are various causes for human error when it comes to producing text. (See [Kuk92] for an overview.) They are often broken down into two categories:

- Physical - eg. typing errors on the keyboard ("*typos*"), such as `‘‘the’’` $\rightarrow$ `‘‘hte’’`
- Cognitive - eg. phonetic errors, such as `‘‘phonetic’’` $\rightarrow$ `‘‘fonetik’’`, where the mistake sounds similar to the intended word, or other confusions such as `‘‘there’’` $\rightarrow$ `‘‘their’’`.

Depending on the application, the errors may be dominated by one type. For example, handwriting recognition would be free from typing errors (but may have other physical pencil errors), numerical entry would be free from phonetic errors, etc... We typically say a word is *misspelled* if it is the result of a cognitive error, and *mistyped* if it is the result of a physical error, but the distinction is not crucial. And though not technically accurate, we use the familiar term *spelling correction* for the task of correcting *any* errors.

The problem isn't limited to these two specific types of error. For example, [Dam64] (40 years ago) acknowledged other sources of error

> *"...where the material to be entered is keypunched or produced on a Flexowriter or other similar device, or where material is sent over electrical circuits and is subject to transmission error."*

The characteristics of the errors are different for each source. For example, typing errors often involve the replacement of characters with those nearby on the keyboard. Spelling errors often involve replacing parts of a word with phonetically similar ones. Errors from an OCR system usually involve mistaking characters for visually similar ones, often unlike human errors (eg. mistaking *"m"* for *"iii"*). See [Kuk92] for details.

Since the entire error process may involve multiple sources of varying degree it is usually difficult to say *which* sources affected a given mistake. Fortunately, one doesn't need to identify the errors in order to correct them. Nevertheless, knowing the characteristics of the error sources will prove useful when making corrections.

## 3 Magnitude and Frequency of Spelling Mistakes

Spelling correction could be thought of as a case of *error correcting codes*, where the redundancy occurs naturally from the sparseness of (English) words in the set of all strings. In classical error correction, the "code book" is designed to allow at most $k$ errors to a string and still be uniquely correctable. Unfortunately in natural language even one error can be uncorrectable, eg. the names `''Tim''` and `''Tom''`. Given the lack of many theoretical guarantees, spelling correction is often forced to make reasonable assumptions in order to make the system as practical as possible.

For example, many of the systems in the literature have a limit on the number or type of errors they can fix. One simplification often made is to assume the correct word will always be at most one character insertion, deletion, substitution, or transposition from the mistake. This assumption greatly simplifies the algorithmic techniques during correction, and makes ranking suggestions easier. This trend was started by [Dam64], who observed that about 80% of spelling errors only had one error. Other studies have observed single-error mistakes to be anywhere from 69% to 95% of all mistakes, depending on the scenario ([Kuk92]).

Another simplifying assumption that can be made is that the first letter will be correct, as this narrows the search space when looking for corrections. This assumption is motivated by evidence that the first letter is more often correct than others: [PZ84] observes only 7.8% of misspellings had the first letter wrong, as opposed to 11.7% and 19.2% for the second and third letters, respectively.

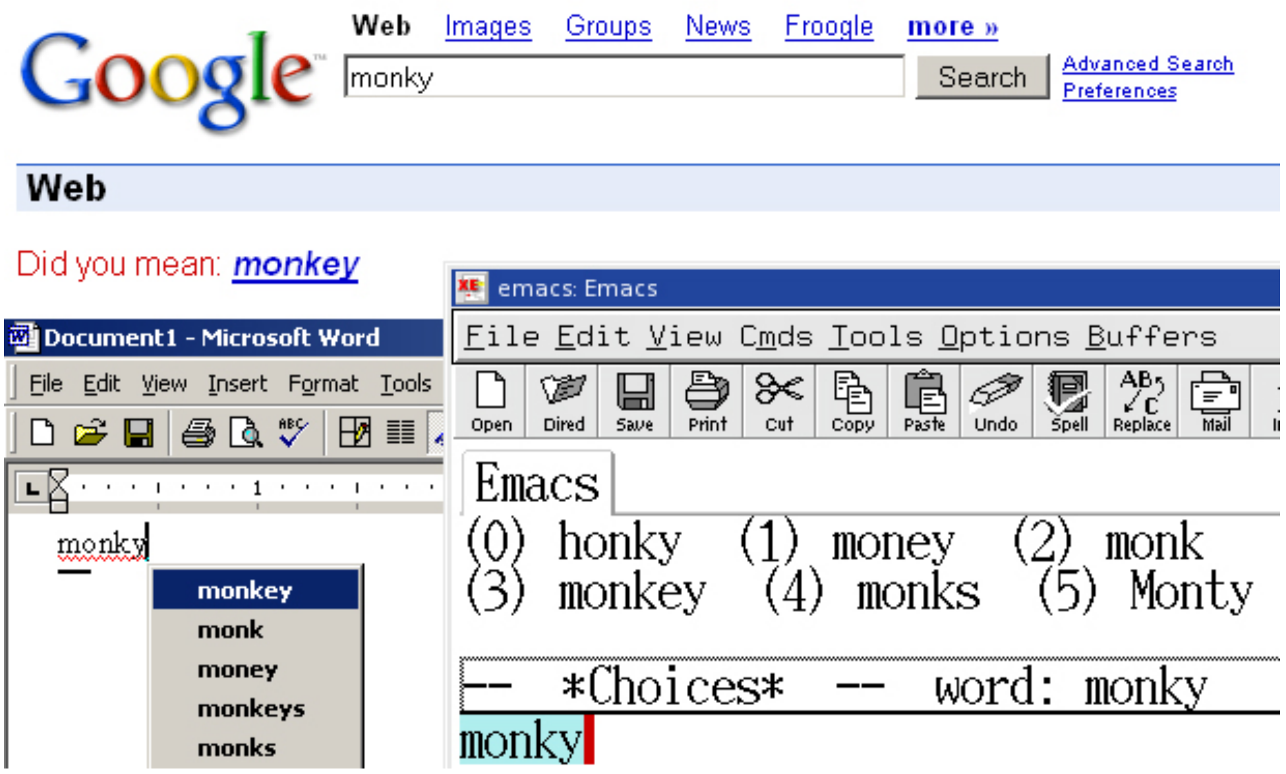
Given assumptions like these, systems in the past have been able to correct most of the errors for which they were designed. [Dam64] was able to correct 95% of single-error mistakes. However, the overall accuracies still have room for improvement. [PZ84] obtained 85%-95% success on mistakes it could have corrected, but only 65%-80% overall.

## 4 General Spelling Correction System

### 4.1 Spell Checking vs. Spelling Correction

The least a system can do is to tell the user which tokens from a piece of text are possible mistakes, *without* making any suggestions on how to fix them. This is the task of *spell checking*. Modern editors like Microsoft Word do this as a background process, marking mistakes with red underlines. Command-line programs like `spell` on Unix simply read in a stream of text, writing out the mistakes.

More helpful is a system that also makes suggestions on how to correct the mistake, so that less input is required from the user. This is the task of *spelling correction*. In Microsoft Word, an underlined mistake can be right-clicked and the correction selected from a menu with one mouse click. In GNU/Linux environments the spell correction program `ispell` (now `aspell`), used by many programs including Emacs, presents a textual menu of suggestions and the user can replace the highlighted mistake by typing a single character corresponding to the desired suggestion in the menu. On search engines like Google or Yahoo a mistaken query also returns a link to a single alternate spelling. (See figure 1.)


Web Images Groups News Froogle more »
monky
Search
Advanced Search
Preferences
Web
Did you mean: monkey
Document1 - Microsoft Word
File Edit View Insert Format Tools
monky
monkey
monk
money
monkeys
monks
emacs: Emacs
File Edit View Cmds Tools Options Buffers
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail
Emacs

```
(0) honky   (1) money   (2) monk
(3) monkey   (4) monks   (5) Monty

-- *Choices* -- word: monky
monky
```

Figure 1: Spelling correction interfaces of modern systems, testing on the mistake ``monky''.

When a spelling correction system is considering possible corrections internally, we refer to them as *candidates*, and the process of producing them as *candidate generation*. Systems usually filter and rank these candidates to present the user with smaller sorted list of *suggestions*.

The least amount of user interaction would occur if a system makes corrections automatically, as a human proof-reader might. Microsoft Word does this for completely unambiguous corrections such as `''onlyy''` $\rightarrow$ `''only''`. But since modern correction systems are not perfect in the general case, the safer method is to present the user with suggestions.

### 4.2 Spell Correction Algorithm Outline

We now present the algorithmic outline that almost all spelling correctors use:

```
SpellCorrect( word $w$ )
    if ( isMistake( $w$ ) )
        $Candidates$ = getCandidates( $w$ )
        $Suggestions$ = filterAndRank( $Candidates$ )
        return $Suggestions$
    else
        return IS_CORRECT
```

First, notice that spell correctors typically do a spell checking step first, to avoid the more costly computation of making suggestions if the word is correct (since most words are). However, we will see certain systems that spell correct *every* word, where the top suggestion is the word itself if it is not a mistake.

Second, we have broken the computation of suggestions into two logical steps: candidate generation and ranking. By *candidates* we mean the set of words that are at least partially considered by the system. Since it would be too costly to consider and rank every word in the language, systems use techniques to eliminate most words a priori. The words left are the candidates. This set could have anywhere from a few, to hundreds or even thousands of words.

However, the goal is to present the user with the smallest set of suggestions as possible. Microsoft Word makes at most five or so ranked suggestions. `aspell` typically makes up to a dozen or so. Google only presents one for a query. Generally, each candidate is given a score internally, and the highest scoring candidates are presented as suggestions.

In some systems, producing suggestions is done in just one step. Nevertheless, we make the distinction for clarity in our presentation of techniques.

## 5 Methods for Recognizing Mistakes

We now focus on the task of spell checking: recognizing when a word is a mistake. Although we are ultimately more interested in making suggestions as well, the issues and techniques addressed here apply to spelling correction, too.

### 5.1 Use of a Dictionary

Typically a set of words (a *dictionary* or *lexicon*) is used to spell check a word: if it is in the set it is correct, otherwise it is a mistake. Recognizing whether a string is in a language is one of the oldest computer science problems. See [Knu81] for a variety of solutions. The straightforward and reasonably efficient methods today include a simple hash table, or perhaps a more compact *trie* of letters.

## 5.2 Space Considerations for Dictionaries

A typical English dictionary contains 20,000 to 100,000 words of average length about 8 characters. A dictionary of this size could be stored completely in memory on today's hardware. However, if we are using extremely large word lists, or otherwise have limited memory, some caching or compression techniques may be necessary.

One easy method is to have a cache of common words in memory, and leave the uncommon words on disk. See [Pet80] for an overview of memory-conserving techniques in spelling correction.

Another space saver is to only store root-forms of words. That is, suffixes and/or prefixes like `''-s''`, `''-ed''`, `''mis-''`, etc... are removed from the dictionary. (See [Pet80] for details.) One problem with this method is that it introduces errors since the *stemming* algorithm that cuts off these affixes is not perfect. The other is that it is English specific: creating a version of this spell checker for another language would require a new stemming algorithm, or may not be possible depending on the language's morphology.

## 5.3 Use of Letter N-grams Instead of a Dictionary

One very space-efficient method for detecting mistakes is to look for words with unusual letter sequences. For example, the last letter-trigrams of the mistake `''standrd''` (`''ndr''` and `''drd''`) are unusual. The statistics for what defines "usual" would be learned from a large corpus of text. The simplest application is to employ "binary trigrams": if a word has any trigrams that did not appear in the training corpus, the word is judged to be a mistake. A more refined version would be to calculate probabilities of each trigram, and compute the probability of the word, with some threshold for when an improbable word is judged as a mistake.

Other than a trigram table - which contains $26^3 = 17576$ entries with most [1] of them 0 - no dictionary is needed. This technique is by no means accurate, and [Kuk92] gives a survey of attempts with mixed results. Letter N-gram analysis is generally considered inadequate for spelling correction. It is more often used in OCR, where errors are more likely to result in unusual letter sequences.

One interesting application of letter trigrams was in [MC75], where N-gram statistics were trained on the document being spell checked! For each token, an *index of peculiarity* was computed - how unlike the rest of the document it was. A sorted list of the most peculiar was then presented to the user, with the hope that all true mistakes were at the top.

## 5.4 Languages with "Infinite" Dictionaries

For languages like English, constructing a list of all words the typical user might write is a feasible attempt. But for *agglutinative* languages like Turkish, a fixed dictionary may not be appropriate. In such languages many affixations are strung together "like beads on a string". As [Ofl96] states

> *"A typical nominal or a verbal root gives rise to thousands of valid forms which never appear in the dictionary."*

[Ofl96] also presents a state-machine like structure to recognize words in languages like these.

[1] 75% according to [Pet80]

## 6 Methods for Generating Suggestions

Once a word is deemed a mistake, the next step is to come up with a set of "possibly intended" words. First, a reasonable assumption is made that the produced mistake looks "similar" to the intended word, as this vastly reduces our search space. Theoretically, we should just consider all words in the dictionary as possible candidates and simply rank them all, presenting the top few to the user. Unfortunately, even for moderately sized dictionaries, considering and ranking every word would make the runtime for processing an entire document extremely long.

Instead, we wish to eliminate the majority of the dictionary (words that are not at all similar to the mistake) a priori. Once this is done, the rest can be ranked by how similar they are to the mistake (along with other factors). Before we can do all this, we need to define what it means for two strings to be "similar".

### 6.1 String Similarity Measures

The similarity measure should reflect the likelihood of transformation under the assumed error process: the similarity between two words $x$ and $y$ should be high if the likelihood that the error process transformed $x$ into $y$ is high, and vice versa. Accordingly many similarity measures model the error process as a sequence of string edit operations, counting the number of operations needed to transform $x$ into $y$. In this case, it would be a *dissimilarity* (or *distance*) measure between strings.

#### 6.1.1 Edit Distance

The *edit distance* is one of the oldest and most widely used measures of how dissimilar two strings are ([Lev66]). It begins by defining a set of edit operations that roughly model the types of errors made by humans:

- insertion - insert a new character somewhere in the string
- deletion - delete a character in the string
- substitution (optional) - replace one character in the string with a different one
- transposition (optional) - swap two adjacent characters

The edit distance between strings $x$ and $y$ is then defined as **the minimum number of edits required to transform $x$ into $y$.**

The first two operations comprise a "complete" set in the sense that any string can be converted to any other using just these two. Substitution could be accomplished by an insertion and a deletion. Transposition could be accomplished by two substitutions.

However, many spelling errors involve substitutions, and swapping two adjacent characters nicely models typing mistakes of that form, as well as phonetic confusions such as `''ie''` and `''ei''`. Hence, these operations are better counted as single operations. The use of all four operations is also called the *Levenshtein distance*, and this is the distance we will use throughout the paper.

The edit distance between two strings $x$ and $y$ can be defined by a recurrence relation:

$$edit(0,0) = \quad 0$$

$$\begin{aligned}
edit(i,0) &= \quad i \\
edit(0,j) &= \quad j \\
edit(i,j) &= \quad min[ \quad edit(i-1,j) \quad + \ 1, & //\text{deletion} \\
& \qquad\qquad edit(i,j-1) \quad + \ 1, & //\text{insertion} \\
& \qquad\qquad edit(i-1,j-1) + \ \delta(x_i,y_j), & //\text{substitution} \\
& \qquad\qquad edit(i-2,j-2) + \ \tau(x_{i-1},x_i,y_{j-1},y_j), & //\text{transposition}
\end{aligned}$$

where $edit(i,j)$ is the edit distance for the substrings $x_{1:i}$ and $y_{1:j}$. $edit(|x|,|y|)$ is the edit distance between $x$ and $y$. $\delta()$ indicates whether an actual substitution occurred (changing to the same character shouldn't count as an edit):

$$\delta(x_i,y_j) \quad = \quad \begin{array}{l} 0 : \text{if } x_i = y_j \\ 1 : \text{otherwise} \end{array}$$

and $\tau(x_{i-1},x_i,y_{j-1},y_j)$ indicates whether an actual transposition occurred, which we count as 1 edit (otherwise it's simply 2 substitutions):

$$\tau(x_{i-1},x_i,y_{j-1},y_j) \quad = \quad \begin{array}{l} 1 : \text{if } (\ x_{i-1} = y_j \text{ AND } y_{j-1} = x_i\ ) \\ 2 : \text{otherwise} \end{array}$$

A recurrence relation lends nicely to dynamic programming, so that the edit distance can be computed in $O(|x| \cdot |y|)$ time. For example, the table of $edit()$ values when $x =$ `car`, and $y =$ `drag` is shown in Figure 2.

| | | c | a | r |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| d | 1 | 1 | 2 | 3 |
| r | 2 | 2 | 2 | 2 |
| a | 3 | 3 | 2 | 2 |
| g | 4 | 4 | 3 | **3** |

Figure 2: Computing the edit distance between ``` ``car'' ``` and ``` ``drag'' ```. The table was filled row at a time, left to right. The final answer is that it takes at least **3** edits to change ``` ``car'' ``` into ``` ``drag'' ```.

We reiterate that an ideal similarity measure is a function whose value is indicative of the likelihood that the error process resulted in that transformation. In particular, if $y_1$ and $y_2$ have the same distance from $x$, they should have been equally likely to be a mistaken output from $x$.

For this to be true of the edit distance it must implicitly assume an error process where insertions occur as frequently as substitutions, and that any character is equally likely to be inserted/substituted, and that the location where the edit occurs is chosen uniformly at random on the string.

It is easy to point out reasons these assumptions fail. We already mentioned that [PZ84] observed the third character was more than twice as likely to carry an error than the first character. Empirically, substitution tends to occur among letters that sound similar or are physically close on the keyboard, and all edits are highly dependent on what the neighboring letters are (see [KCG90] for tables of statistics).

Also, while the edit distance is a symmetric function, the error transformation is inherently a one-way process: the likelihood that someone misses the last letter on ``farm'' producing ``far'' seems greater than the likelihood that someone spuriously types the letter 'm' after ``far'', producing ``farm''.

Despite these inaccuracies, the edit distance is remarkably robust, finding applications in other fields like DNA sequence analysis. Nevertheless, many enhancements and replacements have been explored. (See [ZD96] for a good survey of techniques.)

The simplest modification is the *weighted edit distance*, where instead of unit cost, each edit operation (depending on which characters are involved) is assigned some fixed real value. One example of this is to employ a *phonetic group discount*. [ZD96] uses the following groups of letters:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| a e i o u y | b p | c k q | d t | l r | m n | g j | f p v | s x z | c s z |

A substitution of one character with another of the same group has a cost of 0.5. [ZD96] also modifies the edit distance so that inserting/deleting a letter *after* one of the same group only costs 0.5. Everything else still has a cost of 1.0. Thus, the weighted edit distance between ``cooky'' and ``cookie'' is only 1.0 instead of 2.0 for the unweighted version, whereas ``pick'' and ``pits'' still have a distance of 2.0 under both versions. One could easily adjust the discounts, or the discount groups, perhaps to include keys that are adjacent on the keyboard in the same group.

[ZD96] was in fact interested in the task of retrieving all similar last names given a query string from a phone book. For this application, two strings should be similar if they have similar pronunciations. One measure evaluated was a phonometric method, where character strings are first converted to their pronunciations (a string of phonemes) and then the edit distance is calculated on the phoneme strings. Text-to-pronunciation is a field of its own, and for their work, they used a published algorithm.

### 6.1.2 Soundex Algorithm

Lastly, [ZD96] evaluated the classic Soundex algorithm, patented in 1918. It begins with the following groups of letters:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| a e i o u y h w | b p f v | c g j k q s x z | d t | l | m n | r |

This method then converts any string into a code as follows:

- the first letter of the string is the first letter of the code
- the other letters are converted to numbers using the groups above
- adjacent repeats of numbers are squeezed into one
- 0's are deleted
- at most 3 numbers are used

So Soundex(``Mississippi'') → M0220220110 → M0202010 → M221. Two strings are judged similar if they have the exact same Soundex Code, and judged not similar otherwise.

Of all the methods and variations evaluated, the weighted edit distance presented above performed the best in terms of last name recall/precision.

### 6.1.3 Learning String Similarity Automatically

One criticism of measures like weighted edit distance is that they require a fair amount of human tinkering - eg. choosing discount groups and discount edit costs. This is a fair objection, and work has been done to try to learn weights automatically.

For instance, [YR98] worked on the task of finding the most likely word given a phonetic transcription (phoneme string) of a voice recording. The underlying similarity measure involved a transducer with edit operations like that of the edit distance, where the weights (costs) of each edit were learned automatically. To do this, they made use of 200,000 labeled examples ( phonetic sequence and the intended word ). It is unclear how many examples would be needed or how well this would apply to spelling correction.

[KCG90] presents a system to learn edit probabilities from a corpus of newswire text, which contained approximately 0.05% typos. Specifically, they attempt to learn the following probabilities (for all characters $a$ and $b$):

- P( insert character $b$ after character $a$ )
- P( delete character $b$ after character $a$ )
- P( substitute character $b$ in place of $a$ )
- P( swap adjacent characters $a$ and $b$ )

This was done using an iterative, EM-like approach. The tables of probabilities were initialized uniformly. Their spelling correction system (based on these probabilities) was then used to spell correct all of the mistakes in the corpus. [2] This provided a set of mistake/correction pairs that was used to re-learn the probability tables. This cycle was then iterated. We should note that their system assumed at most one edit in a mistake, which vastly simplifies this scheme.

## 6.2 Generating Candidates

Now we consider the task of taking a mistake and producing a set of candidate words, which hopefully contains the intended word. The typical way to do this is to organize the dictionary in such a way that most of it is never even considered for a given query. For the part of the dictionary that *is* explored, each word is briefly checked to see if it meets some constraints. The words remaining are the candidates.

### 6.2.1 The Case of Single-Error Mistakes

If the simplification is made to only look for corrections that are at most one edit from the mistake, the task is much simpler. Consider the classic system by [Dam64]. Every word in the dictionary is considered, one by one. However, if the lengths of the mistake and the dictionary word differ by more than 1, it can be skipped immediately. Also, a 26-bit "character register" is precomputed for each word, where bit $i$ is set iff the $i^{th}$ letter of the alphabet is in the word. If the character registers of the word and the mistake differ in more than 2 bit locations the word can be skipped. Otherwise, the word is compared to the mistake, one character at a time, allowing at most one mistake along the way. The first word to pass these tests was returned as the correction.

[2] a "mistake" was simply any word rejected by the Unix spell-checker `spell`.

[PZ84] designed a system that sorts the dictionary in a special order so that only a smaller portion of it needs to be considered. A *similarity key* was computed for each word by concatenating the first letter, the unique consonants, and then the unique vowels. For example, `‘‘chemomagnetic’’` has a key of `‘‘chmgnteoai’’`. The dictionary was sorted according to this key, alphabetically. Given a mistake, its key was computed and its (empty) location in the list found. The neighboring (say 50) entries in the list were then considered, making sure they were in fact 1 edit away from the mistake (using a similar tactic as [Dam64]). The intuition behind this similarity key is that the first letter and consonants are more important in judging similarity than the vowels.

However, if the mistake had the wrong first letter, a correction would not be found. To fix this, an *omission key* was also created for each word. It was composed by taking the consonants of a word in the following order: `‘‘kqxzvwybfmgpdhclntsr’’`. This magical order is actually the empirical in-frequency of deleting that character - people rarely forget `k`’s but often forget `r`’s. A second copy of the dictionary was sorted by this key’s order, and it was searched in the case the first search did not return any results.

### 6.2.2 The Case of Multiple-Error Mistakes

The task of determining whether two strings differ by at most 1 error can be computed in linear time by simply walking along both strings allowing one edit if needed. [3] However, the general edit distance takes quadratic time. Even though the strings are typically small, computing the edit distance against an entire dictionary would be too time consuming. [4] One approach then, is to scan the entire dictionary, but quit early if the number of results is large enough, or if the words start to get too dissimilar.

For instance, [EE98] partitions the dictionary into segments based on the word’s first letter and the word’s length. For a given mistake, we consider segments that have the same starting letter, or a similar sounding letter, or a nearby letter on the keyboard, or all other letters, in that order. Within words of a given starting letter, words of the same length were considered, then words of one greater, one shorter, two greater, etc.. were considered in that order. Along the way, the high score of the best match so far is updated, and comparisons of strings can be stopped early if it is clear that it cannot beat the high score.

Lastly, we mention the system of [Ofl96], which constructs a state-machine (really just a *trie* of letters) that only accepts words from the dictionary. However, to generate all candidates at most $k$ errors from the mistake, the state-machine is traversed in parallel, allowing incorrect transitions so long as the total number doesn’t exceed $k$. This method was implemented on a 200,000 word dictionary, and was able to generate candidates with $k = 1$ in 20mS. Alas, this method doesn’t scale well with $k$; for $k = 3$, candidate generation took over 1 second.

Unfortunately there doesn’t seem to be a general-purpose candidate generation algorithm. Each work cited in this paper designed their own method specialized to their purposes. While the task has been solved by practical systems, the general task of finding all words in a list that are “similar” to a query word (that is scalable in the size of the list and the “similarity radius”) appears to be an open problem.

[3] In general, if we assume the edit distance is at most $d$, computing it for two strings of length $n$ takes $O(d \cdot n)$.

[4] Timing experiments by the author indicate the edit distance requires roughly 10 $\mu$ S on average, which on a 100K dictionary would result in 1S of computation.

### 6.3 Ranking Candidates

Ranking candidates typically involves computing a numerical score for each one, but this is not necessarily required. For instance, the system by [Dav62], used to find passenger names, had an ordering of candidates where exact matches came first, exact matches of the Soundex-like code AND the first name initial came second, matches of the code but not the first name initial came third, etc... To rank candidates that were exactly one edit from the mistake, [PZ84] ranked them by which edit it was: deletions and transpositions came before insertions, which came before substitutions.

[EE98] effectively computed a weighted edit distance to each candidate, but discarded candidates that had a bigger distance than the best so far. Also, part of speech analysis was done to discard candidates that violated syntactic constraints with neighboring words.

The seminal paper on candidate scoring was by [KCG90], who combined the probability that a single edit transformed the candidate into the mistake, along with word frequency information about that candidate. Systems like these will be covered in the section on the Noisy Channel Model of Spelling Correction.

## 7 Difficulties for Spelling Correction Systems

### 7.1 Tokenization

Throughout this paper we have thought of a document logically as a sequence of words. In practice, the document is a stream of character bytes, and the words must be *tokenized.* Ideally, each word would be composed of alphabetic characters, separated from other words by spaces. Unfortunately, most punctuation can act as a *word separator* or a *word component* depending on the situation (see Figure 3).

| Punctuation | character | As a word separator | As a word component |
|---|---|---|---|
| , | (comma) | one, two, three | 1,000,000 |
| . | (period) | Jesus wept. | at 5 p.m. |
| ’ | (apostrophe) | ’quoted text’ | didn’t |
| - | (hyphen) | one-of-a-kind | nickel-plated |
| * | (asterisk) | 5 * 5 = 25 | M*A*S*H* |

Figure 3: A sample of punctuation characters that can act as both word separators or word components.

It is easy to think up “if/then/else” rules regarding punctuation, but such rules are never perfect, and can easily grow, becoming difficult to manage. Perhaps considered too unimportant of a task, there doesn’t seem to be much research in this area. But one might surmise that near-perfect tokenization rules (perhaps based on N-grams) could be learned from extremely large corpora such as the Internet.

One particular difficulty in English is the ability of many words to take the suffix `'s`. It is tempting to treat this as a special case and ignore the suffix, only spell correcting the root. For one, it reduces the dictionary size by not having to store this version of each root word. [5] Secondly, it is difficult to even find lexicons that list all words that can possibly

[5]If you count words with punctuation on the inside as separate words, 9% of the 842,000 unique tokens in the 500 million word North American News Corpus ended in `'s`.

have this suffix. While inelegant, this special rule may in fact be the most effective.

The *casing* of words is also important. English words typically come in three forms: lower-case, capitalized, and upper-case. The easiest thing for an implementer is to *case-fold* all lexicon and training corpus words to lower-case. Then, during spell correction, the casing of suggestions is matched to the mistake. For example, `‘‘Aple’’` should be corrected to `‘‘Apple’’` but `‘‘aple’’` to `‘‘apple’’`. However, a useful ability of the spelling corrector might be to correct the casing for words like proper nouns when needed. To do this, proper nouns in the lexicon should be stored in capitalized form so that they are always suggested as such.

## 7.2 Word Boundary Mistakes

Even if tokenization were perfect, user mistakes can result in word boundary problems. Primarily, when a user produces a *run-on* mistake (eg. `‘‘ofthe’’`), an *unintended split* (eg. `‘‘specifi cation’’`), or a *missplit* (eg. `‘‘o fthe’’`), this becomes a complicated matter for a spelling corrector. As stated by [Kuk92]

> *“The difficulty in dealing with run-ons and splits lies in the fact that weakening the word boundary constraint results in a combinatorial explosion of the number of possible word combinations or subdivisions that must be considered.”*

The partial solution, as [EE98] does, is to assume no other mistakes occurred and consider all single space errors. That is, if a word $w$ is a mistake, try:

- for each division of $w$ into $w_1$ and $w_2$, if $w_1$ and $w_2$ are both correct, return this solution.
- concatenate $w$ with the next (or previous) word, and if this results in a correct word return this solution.

When other errors are also involved, this method fails. For instance, [EE98] points out these uncorrected instances:

- `... quite a sop[h isticated one ...`
- `... is a deter miniic statement ...`

Given an $N$-letter word that is a possible run-on, one would have to consider all $2^{N-1}$ partitions, consider all candidates of each word in a given partition, and then have a way to rank all these combinations. Obviously, most partitions and corrections would be unlikely, but navigating efficiently through this search space is a difficult problem. As [Kuk92] sums up

> *“... the general problem of handling errors due to word boundary infractions remains one of the significant unsolved problems in spelling correction research.”*

Since then, there doesn’t seem to have been any progress.

## 7.3 False-positives

For a spell checker looking for mistakes, a *false-positive* is a word marked as a mistake when in fact it is correct. In dictionary based methods, false-positives are words such as proper nouns, special domain jargon, or other uncommon words not found in traditional dictionaries.

Just in terms of vocabulary coverage, [Kuk92] cites a study that observes the overlap between the words in the *Merriam-Webster Dictionary* and an 8 million word corpus of New York Times newswire was only about one-third. Simply put, an insufficient dictionary is the cause of false-positives.

Moreover, mistakes involving valid out-of-dictionary words are *uncorrectable*. The system will either have no suggestions or make *false-corrections*. For example, [PZ84] (using a dictionary of 40,000 words and spell-checking various scientific data sets) observed 5% to 35% of non-dictionary mistakes involved valid words that were not in the dictionary.

## 7.4 Obtaining Large Dictionaries

There is large debate whether larger dictionaries are beneficial overall. This controversy aside, for implementers that wish to use large word lists there are some difficulties. Most free word lists are of moderate size: the `‘‘linux.words’’` file in Linux contains roughly 20,000 words, the CMU pronunciation dictionary contains about 120,000 words including many names.

In order to have a word list that truly covers all domains, it would have to include many names, places, terminology from medicine, biology, etc... One interesting idea is to obtain word lists automatically from large corpora. For example, the North American News Corpus (of about 500 million tokens) contains over 500,000 unique tokens. One of the largest available corpora - the Internet - is a promising place to gather word lists of extreme coverage.

The problem with the Internet, and even “cleaner” corpora such as the news corpus, is that they are rife with mistakes! A simple approach to clean these dirty lexicons is to discard tokens that appear less than a certain number of times. Unfortunately, common mistakes appear just as frequently as other valid words (see Figure 4).

| | | | |
|---|---|---|---|
| recieved | 90 | recieve | 38 |
| survivalists | 90 | subsets | 38 |
| mackovic’s | 90 | panjshir | 38 |
| savr | 90 | angelie’s | 38 |
| trevelyan | 90 | lovato | 38 |

Figure 4: A sample of tokens from the North American News Text, along with their counts. Words in red are clear mistakes a dictionary should not have. Words in blue are valid words a dictionary *should* have. Clearly, word frequency alone is not a perfect indicator of correctness.

Although obtaining large word lists from corpora is attractive, this author considers the automatic construction of extensive (and error-free) word lists from large corpora to be a big open problem for spelling correction.

### 7.5 False-negatives

A *false-negative* is a mistake that is judged to be correct by the spell checker. It's also called a *word-to-word* mistake, since the mistake also happens to be a dictionary word.

False-negatives can be the result of a large dictionary introducing many rare words. For example, the word `''veery''` is most likely a mistyping of `''very''`, but is actually a correct name for a species of small bird.

Other false-negatives involve equally-frequent "confusion pairs", including homophones like `''hear/here''`, `''dessert/desert''`, and `''their/there''`, or other pairs that are only one edit apart, such as `''from/form''` and `''if/of''`. [GS96] used trigrams and other features to correct pairs like these. Essentially, this problem is an instance of *word-sense disambiguation*.

## 8 The Need for Better Techniques

In this section we intend to emphasize that the problems just mentioned are significant, and that the only way to progress toward near-perfect spelling correction is to use large dictionaries and incorporate word frequency information as well as the context of neighboring words.

First, on the issue of false-positives, which was already shown to be statistically significant, we note that a large dictionary is the only way to address it: you cannot know a word is correct unless it is in a dictionary. Hence we argue for extremely large dictionaries.

Unfortunately, using a large dictionary introduces a huge number of problems. Even obtaining one is difficult, as we mentioned. But using one also increases the memory and runtime requirements proportionately. Nevertheless, since hardware will continue to outpace the growth of language, this should soon become a non-issue.

The biggest problem is the increase in false-negatives. [Pet86] investigates the rise in false-negatives as dictionary size increases, and concludes that "word lists used in spelling programs should be kept small." We feel this thinking is timid and outdated. Moreover, the problem of false-negatives is not solved by reducing dictionary size, as there are still confusion pairs such as `''desert/dessert''` that need to be dealt with. Hence better techniques are needed for dealing with false-negatives anyway. And in fact, the task is impossible without the use surrounding context.

Large dictionaries also increase the set of candidates for a given mistake, making it more difficult to get the intended word as the top suggestion. Since larger dictionaries introduce mainly rarer words, the use of word frequency to discount rarer words in the suggestion list looks promising. In addition, context should also come to the aid in ranking candidates.

Correcting multiple-error mistakes also makes ranking more difficult. Consider the mistake `''thermit''` whose candidates include `''thermite''` (one edit away) and `''termite''` (two edits away). Although `''thermite''` is a rarer word, it is more similar to the mistake. How would a system decide between the two? Clearly, we need a method of quantifying the similarity between a mistake and a candidate in such a way that can be used for scoring.

However, **the biggest problem for spelling correction** is the need for a way to **combine** the similarity measures, word statistics, and context information in a structured and formal manner. Work as early as [PZ84] was able to realize that

> *"Two criteria for ranking alternative corrections are the relative frequencies of the target words in the database and the probabilities of the error operations*

*involved.”*

but failed to quantify these factors or combine them together. In the next section we will see a model that addresses many of the above concerns.

## 9 Noisy Channel Model of Spelling Correction

The *Noisy Channel Model of Spelling Correction* formalizes the task of selecting the most likely candidate as an instance of *Bayesian Inference*. All the influences of word frequency, context, and word similarity are required to be computed in probabilistic terms so that they can be easily combined.

### 9.1 Definition

The derivation goes as follows. Let $S$ be a word, phrase, or sentence intended by an author. Let $\widetilde{S}$ be the produced sequence after physical and cognitive errors occur. Our task in correction is to find the most likely sequence $S^*$ given our evidence $\widetilde{S}$. (Hopefully $S^* = S$.) Algebraically,

$$\begin{aligned}
S^* &= arg\max_{S'} P(S'|\widetilde{S}) \\
&= arg\max_{S'} \frac{P(\widetilde{S}|S') \cdot P(S')}{P(\widetilde{S})} && \text{//Using Bayes Rule} \\
&= arg\max_{S'} P(\widetilde{S}|S') \cdot P(S') && \text{//Since } P(\widetilde{S}) \text{ doesn't depend on } S'
\end{aligned}$$

$P(S')$ is the *prior* for candidate $S'$, which for our application we call the *language model* as it is a model of the author's intentions.

$P(\widetilde{S}|S')$ is the *likelihood* that $\widetilde{S}$ is produced when $S'$ was intended. We call this the *error model*.

It is the so-called noisy channel model because it can be thought of as modeling the scenario where the author's intentions go through a noisy channel, resulting in a different distribution at the end (see figure 5).

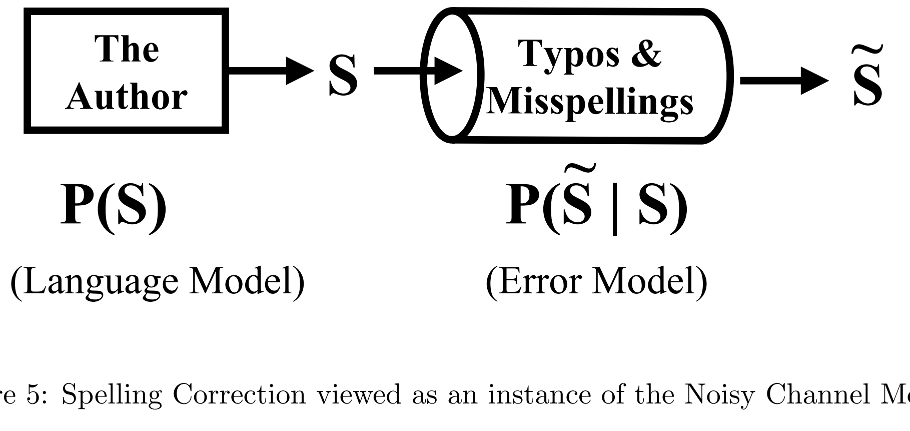


Figure 5: Spelling Correction viewed as an instance of the Noisy Channel Model.

The beauty of this model is that it breaks the problem down into two smaller pieces. All word statistics and context information fall under the language model, and all typographic, phonetic, and other mistakes fall under the error model. Also note that there is no notion of "correct" or "mistake" in this model. If a word is correct, theoretically the most likely candidate should be itself ($S^* = S = \widetilde{S}$).

## 9.2 Founding Work

The concept of a noisy channel was first introduced by Shannon ([Sha48]) and since then has been applied to fields such as voice recognition ([Jel97]). The application to spelling correction was first mentioned in [KCG90] and [MDM91].

In [KCG90], the language model was a unigram model - the probability of a word is simply the normalized frequency of the word in a newswire corpus. The error model was just the probability of the single edit that could have transformed the candidate into the mistake. For example, the candidates for the mistake $\widetilde{S}$ = `''acress''` were ranked as follows:

| $S'$ | $P(S')$ | $P(\widetilde{S}\|S')$ | $P(S')$ * $P(\widetilde{S}\|S')$ |
|---|---|---|---|
| `acres` | $6.5 * 10^{-5}$ | $6.6 * 10^{-5}$ | $4.3 * 10^{-9}$ |
| `actress` | $3.1 * 10^{-5}$ | $1.2 * 10^{-4}$ | $3.6 * 10^{-9}$ |
| `across` | $1.9 * 10^{-4}$ | $9.3 * 10^{-6}$ | $1.8 * 10^{-9}$ |
| ... | ... | ... | ... |

Unfortunately, the correct answer happened to be `''actress''` in this case: "stellar and versatile **acress** whose combination of sass and glamour..." However, a better language model involving neighboring words would probably have given `''actress''` more weight. Overall, their system was able to automatically correct 87% of mistakes.

[MDM91] used a word trigram model borrowed from an IBM voice recognition project. However, the error model was quite simple. It assumed a word is correctly typed with probability $\alpha$. (Empirically, the best value for $\alpha$ was in the range $[0.99, 0.999]$.) Otherwise, given a mistake, the $1 - \alpha$ was divided equally among all candidates that were one edit away from the mistake.

For their experiment, they constructed artificial word-to-word mistakes by taking a correct sentence and replacing one word with a random dictionary word that was one edit away. Their system was able to change the sentence back to its original form 73% of the time. Also, they applied the system to each original sentence - the system should declare that the most likely candidate is itself. It incorrectly changed 1% of the original sentences.

## 9.3 Current Work

Work on designing better language models has been done in many fields, and surprisingly it is difficult to significantly outperform trigrams [Jel93]. Instead, recent work on spelling correction has focused on improving the error model.

Rather than using traditional single character edit operations, the system by [BM00] learns generic substring to substring edits, such as `''ent''` $\rightarrow$ `''ant''` and `''ph''` $\rightarrow$ `''f''`. To do this, a set of 10,000 mistake/correction pairs are used as training to learn both the substrings along with their edit probabilities. Consider the mistake `''fisikle''`, and evaluating the candidate `''physical''`. First, the words are partitioned into small pieces, and aligned:

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| ph | y | s | i | c | al |
| f | i | s | i | c | le |

The probability $P(\widetilde{S}|S')$ is then calculated as

$$P(\texttt{fisicle} \mid \texttt{physical}) = \\ P(\texttt{f|ph}) * P(\texttt{i|y}) * P(\texttt{s|s}) * P(\texttt{i|i}) * P(\texttt{c|c}) * P(\texttt{le|al})$$

Using this error model combined with a trigram language model resulted in over 97% accuracy on a test set of common mistakes.

In a follow-up work, [Tou02] extends this error model to include phonetic errors explicitly. In fact, they use two separate error models. The first is identical to that of [BM00].

The second model begins by taking the mistake/correction pairs and translating each string into a phone sequence. (As a subproject, they build a letter-to-phone pronunciation algorithm to do this.) Then they learn generic phone-subsequence-to-phone-subsequence edit probabilities just as in [BM00]. By mapping strings $S'$ and $\widetilde{S}$ to pronunciations, and then computing the transformation probability based on phones, this results in another error model of the form $P(\widetilde{S}|S')$.

The probabilities of the two error models are then combined log-linearly. When using just a unigram language model, their system of combined error models had an accuracy of 95.6% compared to 94.2% when only using the letter model, and 86.4% when only using the phonetic model.

## 9.4 Conclusions

The Noisy Channel Model has very moderate requirements. It requires that one construct a model of the error process, a model of the generation process for words in the language, that these two models be independent, and that they can be evaluated quantitatively on a probabilistic scale.

It also alleviates many of the difficulties of spelling correction. Although it doesn't address how to enumerate candidates involving word boundary mistakes, e.g. `''toher''` instead of `''to her''`, it does provide a consistent way to rank them. The probability of forgetting a space can be modeled just like any other key so that $P$( `''toher''` | `''to her''` ) can be compared to $P$( `''toher''` | `''other''` ). And most language models can be applied to multiple words so that $P$( `''to her''` ) can be compared to $P$( `''other''` ).

Large dictionaries can be used without fear of introducing too many rare words. Those rare words (like `''veery''`) will have such a small prior that they will not be used by the spelling corrector unless there is overwhelming support from the language model.

## 10 Conclusions and Future Directions

We have given a thorough investigation of the spelling correction problem. Work from 1962 to present has been surveyed, and a broad range of techniques illustrated. We conclude by summarizing the current issues and predicting future directions of research.

One noticeably lacking aspect of the literature is a set of efficient and concise algorithms for candidate generation, similarity measurement, fixing word boundary problems, etc... Each paper cited invents their own methods for these tasks.

As the scope of correction expands to multiple errors, word-to-word mistakes, word boundary mistakes, incorporating context, etc... so does runtime and memory requirements. In a server environment, such as search engines, memory requirements such as storing large trigram statistics may not be a concern. Perhaps the trend will be for clients to upload text to be spell checked on the server, as some email clients do today.

The Noisy Channel Model is a clean and formal model for spelling correction. Systems outside this model have relied on hand-coded rules/formulas and other 'hacks' to do candidate ranking. Regardless of whether this model is used, better language models should no doubt improve spelling correction. Fortunately, language models are being improved by other fields like voice recognition, so spelling correction should be able to benefit from this.

One issue for language models is the need for *clean* corpora. The language model used in the noisy channel model is expected to represent the *source* of words *before* errors occur. But large corpora like the Internet are samples of words *after* errors have occurred; that is, it lets us learn $P(\widetilde{S})$, not $P(S)$.

Unfortunately, even "clean" corpora such as newspaper archives contain a surprisingly high number of errors, and verifying or correcting corpora by hand is not a scalable option. One method might be to use a bootstrap procedure to train an initial system on the raw data, and use that system to automatically spell correct the entire corpus. The new 'clean' corpus could then be used to train the system again. It is difficult to say whether such an approach is feasible with large corpora of billions of words.

Lastly, an interesting potential direction of spelling correction is personalization: adapting the language model to the author or the topic of discourse, or to tune the error model to mistakes often made by that person, or by people of certain cultural backgrounds, etc...

## 11 Acknowledgments

I would like to thank Professor Sanjoy Dasgupta for advising me throughout this project, and always taking the time to entertain my crazy ideas. Also, thanks to Professor Russell Impagliazzo and Professor Amin Vahdat for reviewing my paper and being on my research exam committee. Lastly, I would like to thank the GNU project and the Open Source Community for providing the operating system, editor, compiler, and typesetting system on which this paper was written.

## References

[BM00] Eric Brill and Robert C. Moore. An improved error model for noisy channel spelling correction, 2000.

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.

[Dav62] Leon Davidson. Retrieval of misspelled names in an airlines passenger record system. *Commun. ACM*, pages 169–171, March 1962.

[EE98] Mohammad Ali Elmi and Martha Evens. Spelling correction using context. In *Proceedings of the 17th international conference on Computational linguistics*, pages 360–364. Association for Computational Linguistics, 1998.

[GS96] Andrew R. Golding and Yves Schabes. Combining trigram-based and feature-based methods for context-sensitive spelling correction. In *Proceedings of the 34th conference on Association for Computational Linguistics*, pages 71–78. Association for Computational Linguistics, 1996.

[Jel93] F. Jelinek. Up from trigrams, the struggle for improved language models. In *European Conference on Speech Communication and Technology*, pages 1037–1040, 1993.

[Jel97] Frederick Jelinek. *Statistical Methods for Speech Recognition.* MIT Press, 1997.

[KCG90] M.D. Kernighan, K.W. Church, and W.A. Gale. A spelling correction program based on a noisy channel model. In *In Proceedings of the Thirteenth International Conference on Computational Linguistics*, pages 205–210, 1990.

[Knu81] Donald Ervin Knuth. *The Art of Computer Programming*, volume Volume 3: Searching and Sorting. Addison Wesley, 1981.

[Kuk92] Karen Kukich. Technique for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[MC75] R. Morris and L. L. Cherry. Computer detection of typographical errors. *IEEE Transactions on Professional Communications*, 1975.

[MDM91] E. Mays, F. Damerau, and R.L. Mercer. Context based spelling correction. In *Information Processing And Management*, volume 27(5), pages 517–522, 1991.

[Ofl96] Kemal Oflazer. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Comput. Linguist.*, 22(1):73–89, 1996.

[Pet80] James L. Peterson. Computer programs for detecting and correcting spelling errors. *Commun. ACM*, 23(12):676–687, 1980.

[Pet86] James L. Peterson. A note on undetected typing errors. *Commun. ACM*, 29(7):633–637, 1986.

[PZ84] Joseph J. Pollock and Antonio Zamora. Automatic spelling correction in scientific and scholarly text. *Commun. ACM*, 27(4):358–368, 1984.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[Tou02] Kristina; Moore Robert. Toutanova. Pronunciation modeling for improved spelling correction. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 144–151, 2002.

[YR98] Peter N. Yianilos and Eric Sven Ristad. Learning string edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

[ZD96] Justin Zobel and Philip Dart. Phonetic string matching: lessons from information retrieval. In *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 166–172. ACM Press, 1996.