# Detection of spelling errors in Swedish not using a word list en clair

Rickard Domeij[*] Joachim Hollman[*] Viggo Kann[*]
Numerical Analysis and Computing Science
Royal Institute of Technology
S–100 44 STOCKHOLM
SWEDEN

**Abstract**

We investigate how to construct an efficient method for spelling error detection and correction under the prerequisite of using a word list that is encoded and not possible to decode. Our method is probabilistic and the word list is stored as a Bloom filter. In particular we study how to handle compound words and inflections in Swedish.

**Keywords**: spelling error detection, spelling error correction, Bloom filter

## 1 Introduction

How to automatically detect and correct spelling errors is an old problem. Nowadays, most word processors include some sort of spelling error detection. The traditional way of detecting spelling errors is to use a word list, usually also containing some grammatical information, and to look up every word in the text in the word list (Kukich, 1992).

The main problem with this solution is that if the word list is not large enough, the algorithm will report several correct words as misspelled, because they are not included in the word list. For most natural languages the size of word list needed is too large to fit in the working memory of an ordinary computer. In Swedish this is a big problem, because infinitely many new words can be constructed as compound words.

There is a way to reduce the size of the stored word list by using *Bloom filters* (Bloom, 1970). Then the word list is stored as an array of bits (zeroes and ones), and only two operations are allowed: checking if a specific word is in the word list and adding a new word to the word list. Both operations are extremely fast and the size of the stored data is greatly reduced.

There are two drawbacks to Bloom filters: there is a tiny probability that a word not in the word list is considered to be in the word list, and we cannot store any other information than the words themselves, for example grammatical information.

The word list is stored encoded in a form that is impossible to decode—this is often a prerequisite for commercial distribution. A program that detects exactly the words that are not in the word list can never protect its word list, no matter how it is encoded. This is because it is possible for a modern computer to test, in a few hours, all reasonable combinations of letters and in that way reconstruct the complete word list. This is a crucial advantage of probabilistic spelling error detection methods.

Under the prerequisite of using Bloom filters we have developed a method for finding and correcting misspellings in Swedish texts. The method also works for other languages, similar to Swedish.

[*]Electronic mail: `domeij@nada.kth.se`, `joachim@matematik.su.se`, `viggo@nada.kth.se`. Correspondence should be addressed to Viggo Kann.

In this paper we describe the concept of Bloom filters and how it is possible, in spite of the restrictions of the Bloom filters, to handle inflections, compound words and spelling error correction. We discuss the differences between correcting touch-typed texts and optically scanned texts.

## 2 Swedish word formation

Swedish is a morphologically rich language compared to English. An ordinary verb in Swedish has more than ten different inflectional forms. This makes word listing a heavy task for ordinary computers.

Most words can also be compounded to form a completely new word. For example, the verb *rulla* (roll) can combine with *skridsko* (skate) to form the word *rullskridsko* (roller skate). Since words can combine without limit, it is not even possible to list them. This is a considerable problem for Swedish spell checkers. A great deal of the tiring false alarms that make Swedish spell checkers impractical are compound words.

As the example of Swedish compounding above shows, it is not always possible just to put two words together to form a compound. Stem alteration is often the case, which can mean that the last letter of the initial word stem is deleted or changed, depending (roughly) on what part of speech and inflectional group it belongs to. Between different compound parts an extra -*s*- is often added. However, individual words tend to behave irregularly, thus making compounding hard to describe by general rules.

## 3 Bloom filters

For a long time, the predominant search method has been *hashing*. The basic idea is to assign an integer to every search key. These integers are then used as indexes into a table that holds all the keys. Ideally, there would be a one-to-one correspondence between the integer indexes and the keys, but this is not necessary and is in fact not even desirable in our application. To achieve good results, it is essential that the function which maps search keys to integers can be quickly computed and that the integers are distributed evenly over all possible table indexes.

If the problem at hand is simply a test for membership (e.g., to check if a word belongs to a word list), then Bloom filters (Bloom, 1970) can be used. A Bloom filter is a special kind of hash table, where each entry is either '0' or '1', and where we make repeated hashings into a single table (using different hash functions each time).

A word is added to the table by applying each hash function to the word and entering '1's in the corresponding positions (i.e., the integer indices that the hash functions return).

To check if a word belongs to the word list, you apply the same hash functions and check if all the entries are equal to '1'. If not all entries are equal to '1', then the word was not in the word list.

It can happen that a word gets accepted even if it is not in the word list. The reason is that two different words may have the same *signature*, i.e., '1's in the same positions. Fortunately, the probability for such collisions can easily be adjusted to a specific application. All we have to do is to change the size of the table and the number of hash functions.

Let us compute the probability that a word not in the word list will be accepted by the Bloom filter. Suppose that the word list consists of $n$ words, that the size of the hash table is $m$, and that we use $k$ independent and evenly distributed hash functions. We would like to compute the probability that the values of the $k$ functions all point to entries equal to '1'.

In the hash table $n$ words have been stored, and for each word $k$ entries have been set to '1'. The probability that a specified entry in the table is still '0' after that is

$$\left(1-\frac{1}{m}\right)^{k\cdot n}$$

assuming that the $k \cdot n$ table entry settings were independent. The probability that $k$ random entries in the table all are '1' is

$$f(k) = \left[1 - \left(1 - \frac{1}{m}\right)^{k \cdot n}\right]^k.$$

The minimum of this function is found when

$$f'(k) = 0 \Rightarrow \left(1 - \frac{1}{m}\right)^{k \cdot n} = \frac{1}{2},$$

which means that the hash table is used optimally when it is half-filled with ones. We get

$$k = -\frac{\ln 2}{n \cdot \ln\left(1 - \frac{1}{m}\right)} \approx \ln 2 \cdot \frac{m}{n} \approx 0,69 \cdot \frac{m}{n}$$

and the error probability is

$$f(k) = 2^{-k}.$$

**Example 1** If the word list contains $n = 100\,000$ words and we choose $m = 2\,000\,000$ as the size of the hash table, we should choose

$$k = \ln 2 \cdot \frac{2\,000\,000}{100\,000} \approx 13.9 \approx 14,$$

i.e., we should use 14 hash functions in the Bloom filter. The probability that a random word is accepted is $f(14) \approx 6 \cdot 10^{-5} = 0.006\%$.

## 3.1 An example of hash functions

Let $c_j$ be the ASCII-value[1] associated with the $j^{\text{th}}$ character in the word $w$, and let $|w|$ denote the total number of characters in $w$. For some chosen prime $p_i$ we compute the hash value[2] $h_i(w)$ as

$$h_i(w) = \sum_{j=1}^{|w|} 2^{(j-1)\cdot 7} c_j \bmod p_i.$$

We should say that there are various ways to speed up the computation of $h_i(w)$, and that an efficient implementation has an apparent effect on the overall performance of our method. One could for instance compute $1/p_i$ once, and use multiplication instead of a straightforward, but costly, remainder taking.

Now, let us take a look at a concrete example. Assume that we want to check if the word *test* is in the word list, and that we have chosen the primes $p_1, \ldots, p_{14}$ to be the 14 largest primes less than 2 000 000, i.e., $p_1 = 1999993$, $p_2 = 1999979, \ldots$, and $p_{14} = 1999771$.

**Example 2** We compute $h_i(\textit{test})$, for $i = 1, \ldots, 14$. The ASCII-values corresponding to the word *test* are $c_1 = 116$, $c_2 = 101$, $c_3 = 115$, and $c_4 = 116$. Applying the above formula for $h_i(w)$ gives us

$$\begin{aligned}
h_1(\textit{test}) &= (2^{0\cdot 7} \cdot 116 + 2^{1\cdot 7} \cdot 101 + 2^{2\cdot 7} \cdot 115 + 2^{3\cdot 7} \cdot 116) \bmod 1999993 = 1167690 \\
h_2(\textit{test}) &= (2^{0\cdot 7} \cdot 116 + 2^{1\cdot 7} \cdot 101 + 2^{2\cdot 7} \cdot 115 + 2^{3\cdot 7} \cdot 116) \bmod 1999979 = 1169398 \\
&\vdots \\
h_{14}(\textit{test}) &= (2^{0\cdot 7} \cdot 116 + 2^{1\cdot 7} \cdot 101 + 2^{2\cdot 7} \cdot 115 + 2^{3\cdot 7} \cdot 116) \bmod 1999771 = 1194774
\end{aligned}$$

The word *test* is accepted if, and only if, the hash table entries 1167690, 1169398, ..., and 1194774 are all equal to '1'.

[1] You can of course use character set maps other than ASCII.
[2] This is just an example of a hash function, any good hash function will do.

## 4 Compounding and inflection

In our program, compounding and inflection are handled by an algorithm that uses a list of ending rules together with three different word lists.

1. the *exception list*, containing words that cannot be part of a compound at all,
2. the *last part list*, containing words that can end a compound or be an independent word,
3. the *first part list*, containing altered word stems that can form the first or middle part of a compound.

Inflection is handled in a straightforward but unconventional way. We are trying a heuristical method to reduce the number of word forms listed, and ensure that all forms of a word is represented. The *last part list* presented above does not actually contain all inflectional word forms. It only contains the basic word forms needed to infer the existence of the rest from ending rules.

Both basic word forms and altered word stems are (semi-) automatically constructed from a machine readable dictionary with inflectional and compound information.

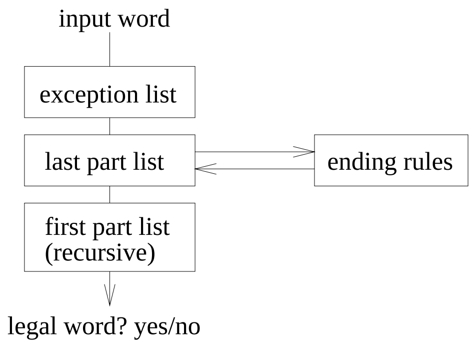

Figure 1: Look-up scheme for handling compounding and inflection.

When a word is checked, the algorithm consults the lists in the order illustrated in Figure 1. In the trivial case, the input word is found directly in the *exception list* or the *last part list*. If the input word is a compound, only its last part is confirmed in the *last part list*. Then the *first part list* is looked up to acknowledge its first part. If the compound has more parts than two, a recursive consultation is performed. The algorithm optionally inserts an extra -*s*- between compound parts, to account for the fact that an extra -*s*- is generally inserted between the second and third compound parts.

The ending rule component is only consulted if an input word cannot be found neither in the *exception list* nor the *last part list*. If the last part of the input word matches a rule-ending, it is considered a legal ending under the condition that the related basic inflectional forms are in the *last part list*. In this way, only three noun forms, out of normally eight, must be stored in the *last part list*. The other noun forms are inferred by ending rules.

**Example 3** The word *docka* (doll) belongs to the first inflectional noun class in Swedish, and has the following inflectional forms:

| | |
|---|---|
| *docka* | (doll) |
| *dockan* | (the doll) |
| *dockor* | (dolls) |
| *dockorna* | (the dolls) |
| *dockas* | (doll's) |
| *dockans* | (the doll's) |
| *dockors* | (dolls') |
| *dockornas* | (the dolls') |

For this ending class only *docka*, *dockan* and *dockor* are put in the last part list. We construct the following ending rules from which the other five forms can be inferred:

| | | |
|---|---|---|
| *-orna* | $\leftarrow$ | *-a*, *-an*, *-or* |
| *-as* | $\leftarrow$ | *-a*, *-an*, *-or* |
| *-ans* | $\leftarrow$ | *-a*, *-an*, *-or* |
| *-ors* | $\leftarrow$ | *-a*, *-an*, *-or* |
| *-ornas* | $\leftarrow$ | *-a*, *-an*, *-or* |

Consider the input word *porslinsdockorna* (porslin=porcelain). The input word cannot be found in the *exception list* nor the *last part list*. Therefore the ending rules are consulted. The first rule above is to be read (somewhat simplified) like this: If the words *dock-a*, *dock-an* and *dock-or* are in the *last part list*, then the word *dock-orna* is a legal word.

Finally the *first part list* is consulted. There the first part of the compound (*porslins-*) is found, thus confirming the legality of the input word.

Our handling of inflections is a possible source of error. For example, the non-existing word *dekorna* can be constructed using the rule above since the words *deka* (degenerate), *dekan* (dean) and *dekor* (décor) all exist in Swedish. It is important to design the rules in such a way that the number of incorrect words that can be constructed is minimized. There are different ways to obtain better rules. We can include a new suffix on the right hand side of the rule, and at the same time expand the word list with the corresponding inflectional word forms. Another way is to substitute a new suffix for a suffix on the right hand side. A third method is to include a *negated suffix* which works in the following way. If the negated suffix $\mathcal{S}$ is included, and a word exists in the word list with the suffix $\mathcal{S}$, then the rule cannot be applied to that word.

In order to compare different variants of ending rules we generate all possible words that can be constructed from a specific rule. Using the rule in the example above, 1532 words can be generated, and only two of them are incorrect. Thus, the error is $2/1532 \approx 0.0013$.

## 5 Spelling error correction

Many studies, see for example Damerau (1964) and Peterson (1986), show that four common mistakes cause 80 to 90 percent of all typing errors:

1. transposition of two adjacent letters,
2. one extra letter,
3. one missing letter, and
4. one wrong letter.

A method that has proven to be useful for generating spelling correction suggestions is to generate all words that correspond to these four types of mistakes, and see which are correct words.

Words that are generated in this way are said to lie at distance of one from the original word. If there are no correct words within this distance, one could continue the search by increasing the distance by one at each step, but this is of course a very expensive process. An alternative method is described by Du and Chang (1992).

This metric is well suited for touch-typed text but other metrics should be used for texts entered in other ways. For instance, hand-written text, and texts that have been entered using OCR-techniques, see Takahashi et al. (1990) and the section below, are likely to contain different types of errors.

A problem with the probabilistic method is that when we generate many suggestions for a misspelled word there is a slight possibility that an incorrect word may slip in. It is however possible to reduce such errors to a minimum by introducing a *graphotactical table* as suggested by Mullin and Margoliash (1990). This table holds all allowed *n-grams*, i.e., combinations of $n$

letters, for some prespecified limit $n$. We have chosen $n = 4$ and we store the graphotactical table using one bit for every possible 4-gram, '1' if there is a Swedish word that contains the 4-gram and '0' otherwise. A word is accepted as correct only if all its 4-grams appear in the table. In Swedish only a small subset of the $n$-grams can appear at the beginning of a word, and likewise only a small subset can appear at the end of a word. Therefore we consider the beginning and end of the word as special letters in the $n$-grams. A graphotactical table for Swedish constructed in this way will be filled to about 8 percent.

The reasonableness of the generated words is checked both against the Bloom filter and the graphotactical table. The words that pass both tests will be suggested as corrections.

**Example 4** Consider the misspelling *strutn*. Generate all words within distance one from this word, check the words using the graphotactical table and using the Bloom filter. We will show below how many words that are left after each stage in this process.

1. Transpose two adjacent letters. 5 generated words (*tsrutn*, *srtutn*, *sturtn*, *strtun*, *strunt*). After checking the graphotactical table only *strunt* is left, which will also pass the Bloom filter.
2. Take away one letter. 6 generated words (*trutn*, *srutn*, *stutn*, *strtn*, *strun*, *strut*). After checking the graphotactical table only *strut* is left, which will also pass the Bloom filter.
3. Insert one letter. $7 \cdot 29 = 203$ generated words (7 places to insert one letter and 29 letters in the Swedish alphabet). After checking the graphotactical table 6 words are left (*strutan*, *struten*, *strutin*, *struton*, *strutna*, *strutne*). Only *struten* will pass the Bloom filter.
4. Replace one letter. $6 \cdot 28 = 168$ generated words (6 letters to replace and 28 letters to replace with). After checking the graphotactical table 13 words are left. Only one (*struts*) will pass the Bloom filter.

Thus four suggestions will be presented: *strunt*, *strut*, *struten*, and *struts*, which are all correct Swedish words.

For a misspelled word of $b$ letters we generate $59b + 28$ words that must be checked. For $b = 10$ we thus must check 618 words. If the misspelling itself introduces a 4-gram that is not in the graphotactical table, then the number of words that have to be checked will be reduced to a number smaller than 208, independent of $b$.

One should note that the graphotactical table has to be updated if we allow the user to add her own words; fortunately, this is easy.

In earlier studies of automatic spelling correction, see for instance Takahashi et al. (1990), it has been considered impractical to use word lists larger than about 10 000 words. Using our methods, it is possible to have extremely large word lists without sacrificing speed.

## 6 Correction in optically scanned documents

Correction in connection with OCR is in many ways different from the ordinary spelling correction described above. Not only are we faced with typing errors, but also errors due to imperfections in the text recognition device used. Even a high quality system with a *character* recognition accuracy rate as high as 99% may result in a mere 95% *word* recognition accuracy rate, because one error per 100 characters equates to roughly one error per 20 words, assuming five-character words.

In an optically scanned document we can expect similar looking characters, or groups of characters, such as: 'O'-'0', 'I'-'1'-'l', '*A*'-'.4', and 'a'-'å'-'ä'-'á'-'à', to cause problems. This is common source of error, especially in a language such as Swedish where 'å', 'ä', and 'ö' are very common "real" letters, i.e., not simply 'a', and 'o' with diacritical marks. Our preliminary results suggest that roughly half of the errors in optically scanned Swedish texts are of this type.

It is natural to choose a metric, i.e., measure of distance between words, different from the one used for (directly) touch-typed texts. In contrast to the usual minimum edit distance, noninteger distances are used here.

Our earlier remarks suggest that this metric depends both on the shape of the characters, and the language ($n$-gram frequencies). As a step toward fully automatic word correction, or at least in order to help the user of an interactive program, the potential corrections should be ordered by increasing distance from the misspelled word. At the moment, we consider this ranking of candidates to be the most interesting practical problem. The reason for this is that in nearly all cases the correct word is to be found among the candidates, so the real problem is to pick the right candidate. We are currently investigating techniques along the lines of Kernighan, Church and Gale (1990, 1991).

## 7 Retrieving the word list

Any spelling error detection program's word list can be retrieved using the following algorithm.

Generate all possible combinations of letters (using the graphotactical table to throw away impossible words) and input them to the spelling error detection program. Note which words the program accepts. These words form the word list.

If the spelling error detection is exact, we have retrieved the word list exactly, but if it is probabilistic, we have got a word list that contains some errors.

If we use the algorithm of our spelling error detection program, we will get about 2% nonsense words, which will make the word list useless for others.

This error rate should not be confused with the probability that a misspelled word is accepted by the Bloom filter, which is 0.006% in our program.

## 8 Performance of our method

Here are some notes on the performance of the current implementation of our method. The computer used is a Sun Sparc station ELC, a Unix machine comparable with a fast 486 PC.

- Speed:
  - looking up words in the *exception list* and the *last part list* only: 2500 words/sec,
  - general spelling detection (including compounding and inflection): 700 words/sec,
  - spelling error correction: 20 words/sec.
- Memory requirements:
  - first part list 100 Kbyte (about 40 000 words),
  - last part list 250 Kbyte (about 100 000 words),
  - exception list 25 Kbyte (about 10 000 words),
  - graphotactical table 100 Kbyte (4-grams, 29 letters in alphabet),
  - ending rules 10 Kbyte (about 600 rules).

## References

Bloom, B. H. (1970), "Space/time trade-offs in hash coding with allowable errors". *Communications of the ACM* 13, 422–426.

Church, K. W., Gale, W. A. (1991), "Probability scoring for spelling correction". *Stat. Comput.* 1, 93–103.

Damerau, F. J. (1964), "A technique for computer detection and correction of spelling errors". *Communications of the ACM* 7, 171–176.

Du, M. W., Chang, S. C. (1992), "A model and a fast algorithm for multiple errors spelling correction". *Acta Informatica* 29, 281–302.

Kernighan, N. D., Church, K. W., Gale, W. A. (1990), "A spelling correction program based on a noisy channel model". In: Karlgren, H. (Ed.), *COLING-90, The 13th International Conference on Computational Linguistics, Helsinki, Finland*, volume 2, 205–210.

Kukich, K. (1992), "Techniques for automatically correcting words in text". *ACM Computing Surveys* 24, 377–439.

Mullin, J. K., Margoliash, D. J. (1990), "A tale of three spelling checkers". *Software–Practice and Experience* 20, 625–630.

Peterson, J. L. (1986), "A note on undetected typing errors". *Communications of the ACM* 29, 633–637.

Takahashi, H., Itoh, N., Amano, T., Yamashita, A. (1990), "A spelling correction method and its application to an OCR system". *Pattern Recognition* 23, 363–377.