Implementation aspects and applications of a spelling correction algorithm

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

7th May 1998

Abstract

A method for detecting and correcting spelling errors in Swedish text was presented by Domeij, Hollman and Kann (1994). The objectives were to perform very fast detection and correction of errors and to use a full size word list. Our implementation of the method as a C program is called STAVA. We have further refined this method and implemented ranking of corrections using word frequencies and editing distance. We also describe how the method can be used in several applications, for example when extending a part-of-speech lexicon, tagging unknown words, stemming and correcting search questions in information retrieval.

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.

*E-mail: viggo@nada.kth.se, domeij@nada.kth.se, joachim@nada.kth.se, d91-mti@nada.kth.se, URL to the project: http://www.nada.kth.se/theory/projects/swedish.html.

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, see section 7. Thus the first drawback of Bloom filters in fact is a crucial advantage, since it makes it possible to protect the word list. In section 4 we show how to work around the second drawback of Bloom filters, by storing word frequency information together with the words.

We have developed a method for finding and correcting misspellings in Swedish texts using Bloom filters. In this paper we describe the method and our implementation of it, called STAVA (Kann, 1998). Especially we concentrate on how the spelling error correction suggestions are ranked using quantitative linguistic methods. We also mention several applications of our methods, besides spelling error detection and correction. Examples of applications that we have studied are extension and creation of a part-of-speech lexicon, tagging of unknown words, hyphenation of solid compounds, stemming and correction of search questions in information retrieval.

2 Preliminaries

2.1 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.

2.2 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.

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. Then the probability that a word not in the word list will be accepted by the Bloom filter is

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

as shown in Domeij, Hollman and Kann (1994). The minimum of this function is $f(k) = 2^{-k}$ which is reached when

$$k = -\frac{\ln 2}{n \cdot \ln\left(1 - \frac{1}{m}\right)} \approx \ln 2 \cdot \frac{m}{n} \approx 0.69 \cdot \frac{m}{n}.$$

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 $k = \ln 2 \cdot 2\,000\,000/100\,000 \approx 14$, 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 Compounding and inflection

3.1 Basic structure of the word recognition

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

1. the *individual word 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 are 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 suffix rules.

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

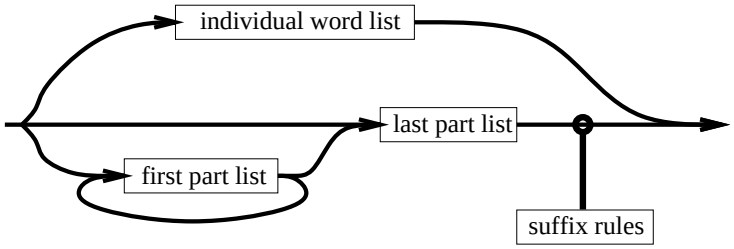

Figure 1: Look-up scheme for handling of compounding and inflection.

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 *individual word 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 *individual word 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 suffix rules.

Example 2 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 suffix rules from which the other five forms can be inferred:

$$
\begin{array}{lcl}
\textit{-orna} & \leftarrow & \textit{-a, -an, -or} \\
\textit{-as} & \leftarrow & \textit{-a, -an, -or} \\
\textit{-ans} & \leftarrow & \textit{-a, -an, -or} \\
\textit{-ors} & \leftarrow & \textit{-a, -an, -or} \\
\textit{-ornas} & \leftarrow & \textit{-a, -an, -or}
\end{array}
$$

Consider the input word *porslinsdockorna* (porslin=porcelain). The input word cannot be found in the *individual word list* nor the *last part list*. Therefore the suffix 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.

3.2 Improved expressiveness of suffix rules

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 our syntax for suffix rules we precede the negated suffix by ~.

Example 3 The rule

$$\textit{-samma} \quad \leftarrow \quad \textit{-sam, ~samen}$$

says for example that the plural form *varsamma* of *varsam* (careful) is a correct word since *varsam* but not *varsamen* is in the last part list. The negated suffix is added so that STAVA should not accept *balsamma* of the reason that *balsam* (balsam) exists. The definite form *balsamen* is namely also in the last part list.

In order to make the suffix rules more expressive we can state that a rule only should be applicable when the suffix is preceded by a certain letter or if it is *not* preceded by a certain letter. We use standard Unix regular expression syntax for this.

Example 4 Consider the following two rules.

$$
\begin{array}{lcl}
\textit{[uå]-rna} & \leftarrow & \textit{-}\varepsilon\textit{, -n, -r, ~dde, ~ra} \\
\textit{[\^{}sxz]-s} & \leftarrow & \textit{-}\varepsilon\textit{, -en, -er}
\end{array}
$$

The first rule accepts the suffix *-rna* only if it is preceded by either *u* or *å*. Thus *basturna* (the saunas) is accepted but not *vararna*.

The second rule accepts the genitive suffix *-s* when it is not preceded by *s*, *x* or *z*. This means that *films* (film's) is accepted but not *sfinxs*.

In order to compare different variants of suffix rules we generate all possible words that can be constructed from a specific rule. Using the *-orna* rule in example 2, 1 592 words can be generated, and only two of them are incorrect (*dekorna* and *traktorna*). Thus, the error is $2/1\,592 \approx 0.13\%$.

3.3 Exception list

It is of course unsatisfactory that a few non-existing words (like *dekorna*) are accepted by the algorithm. If we can identify these words we can avoid this problem by putting them in an *exception list*. This word list should contain all words that are wrongly accepted by the algorithm presented in section 3.1 and should be searched before any of the other word lists. If the exception list is stored as a Bloom filter using same hash functions as the individual word list and the last part list, the only extra work when checking a word will be to look at a few (at most k but most often just one or two) positions in an array of bits.

Another advantage of introducing an exception list is that we may include words in the last part list that only exist as last elements and not as whole words. In Swedish there are a couple of such words, for example *mässig* that is a very common last element in compounds like *affärsmässig* (businesslike) but cannot be an individual word. If such words are added both to the last part list and the exception list, we obtain the desired result.

In some cases there are common misspellings that coincide with very uncommon inflectional forms of other words. An Swedish example of this is the misspelling *parantes* of *parentes* (parenthesis). Unfortunately *parantes* is the masculine genitive inflection of *parant* (stylish). This means that the spelling error detector will accept *parantes* even if it almost surely is a misspelling. The solution to this problem is to add the word to the exception list.

3.4 Suffix rules without errors

The reason that we have chosen the type of suffix rules described in section 3.1 is that the word lists that we have access to do not contain any paradigm information for the words. Thus we have to rely on that certain inflectional forms of each word in each paradigm are included in the word list. This leads to the problems with overgeneration that were described above.

If we had access to a word list where each word's paradigm is marked we could use another and completely safe type of suffix rules. In the right hand side of each rule we just have one suffix (for the primary form) and a code for the word's paradigm. In the word list (last part list) we only store the primary form of each word and attach the code of the paradigm to the end of the word.

Example 5 Suppose that the paradigm of the first inflectional noun class in Swedish has the code 17. Then we include *docka17* in the last part list and write the suffix rule for the definite plural form as

$$\textit{-orna} \quad \leftarrow \quad \textit{-a17}$$

4 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: transposition of two adjacent letters, one

extra letter, one missing letter, and 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 a distance of one from the original word.

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 7 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 6 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.

5 Spelling correction with ranking

5.1 The need for ranking

When an interactive spell checker finds a spelling error, it usually asks the user if and how she wants to correct the error. A few spelling corrections are then presented. The algorithm suggested in section 4 will find some possible corrections at a distance of one from the original word. If there are no words at a distance of one, it can compute the words that have a distance of two from the original word instead. In any case there might be a number of suggestions, and ideally they should be ranked so that the most probable correction is given as the first alternative, the second most probable correction as the second alternative and so on. If the algorithm makes a correct guess, it is easy for the user to make the change.

In some cases (for example in OCR and in spelling correction for information retrieval, see section 8.6) there might be need for fully automatic spelling correction, i.e., the program corrects the errors without asking the user first. In this case it is of course very important that the algorithm with high probability makes the right choice among the possible corrections.

A third possibility is a semi-interactive spelling correction that reports corrections when it is clear which word the user intended to write, and asks the user when there is no single correction that is significantly more probable than the others. Then the algorithm must be able not just to *rank* the suggestions but to give them a *probability*.

We have studied the ranking problem under the same objectives as before, i.e. the algorithm should be fast and the full size word list is encoded as a Bloom filter. We found that the best result was obtained when we used both a refined editing distance and word frequency information for ranking the corrections (Tillenius, 1996).

Each correction suggestion is given a *penalty*, which is a number that tells how (un)probable this word is as the correction of the misspelled word. The penalty is a combination of an editing distance penalty and a word frequency penalty.

5.2 Refined editing distance

The editing distance penalty is dependent both on the edit operation and the letters surrounding the place of the operation. For insertion the penalty is dependent on the letter inserted and the letter following it. For deletion the penalty is dependent on the letter deleted and the letter following it. For replacement the penalty is dependent on the new letter and the letter it replaces. For transposing the penalty is dependent on the two letters that are transposed. It should also be an extra penalty for changing the first letter in a word since it is uncommon for the first letter to be wrong.

These rules can correct all of the normal keyboard typing errors and make it possible to code their probabilities (e.g. an *a* is more often mistyped as an *s* than as a *p* on a normal keyboard). The rules are also powerful enough to correct some phonetic errors. Since the correlation between spelling and pronunciation is high in Swedish, these rules work quite well for most common Swedish phonetic errors. An example is the misspelling *gort* of *gjort* (made) that is quite common since the consonant [*j*] is spelled *g* more often than *gj*.

The insertion and deletion rules will also take care of doubling and undoubling of consonants (e.g. *spel* $\leftrightarrow$ *spell*, *tik* $\leftrightarrow$ *tick*), which are very common types of errors in Swedish.

The penalties can be generated rather easy by collecting statistics of real spelling and typing errors.

5.3 Word frequency

The word frequency penalty depends on how common the word is in the Swedish language (ideally taken over the text type that the user currently is writing). More common words give a lower penalty. We chose to divide the words into 10 frequency classes named *A, B,...,J*. The word frequencies were stored in a separate Bloom filter where each word was concatenated with the letter corresponding to its frequency class. For example the very common word *och* (and) is in frequence class *A* and is thus stored in the Bloom filter as *ochA*. In this way the frequency class of a word can be found by at most 10 look-ups in the Bloom filter.

5.4 Evaluation of our ranking

An evaluation of the ranking method on 729 misspelled words shows that it finds the correct correction in 60% of the cases, see table 1. This is very good, especially taking into consideration that only 78% of the corrections were included in the word list of the program.

Method	1	2	3
none	204 (28%)	71 (10%)	16 (2%)
word freq.	356 (49%)	42 (6%)	26 (4%)
edit dist.	388 (53%)	55 (8%)	16 (2%)
edit dist.+word freq.	440 (60%)	28 (4%)	10 (1%)

Table 1: Performance of different spelling correction methods tested on 729 misspelled words. The columns 1, 2 and 3 tell whether the correct word was the first, second or third suggestion. None means that no ranking was performed, the suggestions were presented in the order they were generated.

We also tried to use word bigrams to rank the suggestions, but this was not successful. The reason was that most correct bigrams were not included in the bigram database (containing 200 000 bigrams) that we used. Word bigrams might work better on tests with a smaller vocabulary. We did not try to use word class tag bigrams, which perhaps would improve the ranking if word tags are available (see also section 8.3).

6 Our implementation: STAVA

We have implemented the algorithms for spelling error detection, correction and ranking that we have described as a C program of 4 000 lines. The program is called STAVA. Documentation and a test version of STAVA are available on the web (Kann, 1998).

In the following we will describe and discuss some implementation aspects.

6.1 Word lists and suffix rules

We have used many sources of Swedish words for STAVA. The main source is the word list of the Swedish Academy (1986) with 120 000 words and information about inflections. For the word frequency list we have used a source consisting of 200 000 words collected from a newspaper corpus of 1 000 000 words composed by Språkdata at the University of Gothenburg.

The last part list consists of about 100 000 words, the first part list of about 25 000 words, the individual word list of about 1 000 words and the exception list of about 200 words.

There are about 1 000 suffix rules in STAVA. When constructing the rules we have used the Swedish morphology as described by Hellberg (1978). The rules are sorted by the suffix on the left hand side reversed (from right to left). This means that we can use binary search when looking for rules that match a given word. About 500 000 words can be constructed from the suffix rules using the last part list.

When suffix rules are matched against a word it often happens that the same word has to be looked up in the last part list several times. In order to minimize the number of look-ups we have a special cache that remembers the last look-ups and their results.

6.2 Optimization of the hash functions

Every hash function in the Bloom filters has the following basic structure, where c_j is the ASCII-value[1] associated with the j^{th} character in the word w, $|w|$ is the total number of characters in w, and p_i is a prime smaller than the size of the hash table.

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

The main part of the execution time (more than 80%) is spent on computing the hash functions. Therefore it is very important to speed up the computation of $h_i(w)$. First we noted that the most time-consuming operation is the mod computation, since the remainder taking hides a division. We tried to do get rid of the division by precomputing $1/p_i$ once for all and using floating number multiplication instead of remainder taking. This improved the total running time by a factor of two.

Next improvement was done by performing mod once per hashing instead of once for each character. This is possible without overflow for short words, but if the program is run on the same computer as the Bloom filter is built we can in fact forget about overflow–the important thing is that the computed hash value is the same each time. This improved the running time by another factor of two.

Now we wanted to get rid of the mod operation completely. If we choose the hash table size as an exponent of 2, the mod operation can be performed by a simple and extremely fast bit mask. This would destroy the even distribution of the above hash function, so we had to change to a hash function that mixes all the bits of the hash value so that taking just the last bits still gives an even distribution. For this we used a hash function constructed by Jenkins (1997).

Finally we observed that the same hash functions (mod different numbers) are computed twice, since a word is searched both in the individual word list and in the last part list. When we had changed the program so that we reused earlier computed hash values we had made a total optimization by a factor of ten with respect to the unoptimized program.

6.3 Performance of our method

Here are some notes on the performance of the current implementation of our method. The computer used is a Sun Sparcstation 10, a Unix machine comparable to a Pentium PC.

[1] You can of course use character set maps other than ASCII.

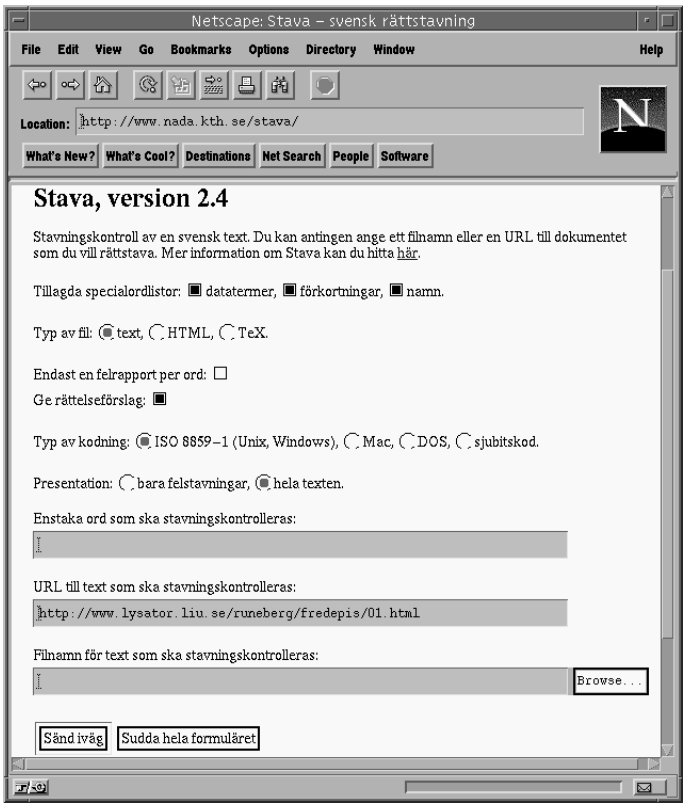

Figure 2: Web interface for Stava.

- looking up words in the *individual word list* and the *last part list* only: 80 000 words/sec,
- general spelling detection (including compounding and inflection): 10 000 words/sec,
- spelling error correction and ranking: 100 words/sec.

6.4 User interface

Although the user interface of the spell checker has not been a major part of the project, we have developed four different interfaces and an API (programming interface).

1. **Unix command line interface**
   The original user interface to STAVA was modeled on the Unix standard utility Spell. On the command line a text file name is given and then it is spell checked and the misspelled words are output. If the option `-r` is given, spelling correction suggestions are also output. There exist several other options that control the behaviour of the spelling correction (Kann, 1998).

2. **Emacs interface**
   We have written a simple interface for the editor Emacs. A single word or a whole buffer (file) can be interactively spell checked. When a misspelling is found the user has the possibility to change it.

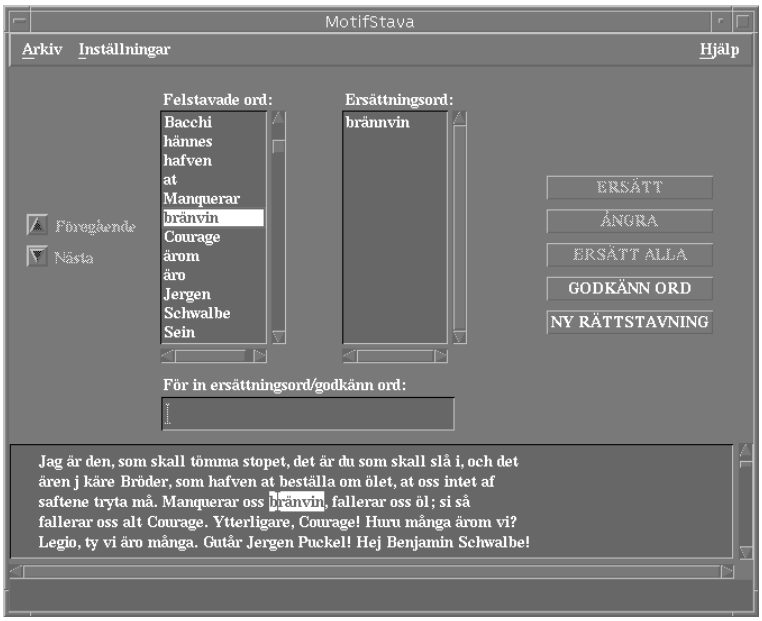

Figure 3: X graphical user interface for Stava.

3. **Web interface**
   Our web interface (Kann, 1998) can spell check words given in the web form, a text file on the user's computer or a web page anywhere in the world, see figure 2. The result may either be a list of misspellings (maybe with correction suggestions) or the whole text where the misspelled words are blinking. If you click on a misspelled word you will see the correction suggestions.

4. **Graphical user interface for X**
   A computer science student has designed and implemented a graphical user interface for the X window system, running on both Solaris and Linux (Johansson, 1997). The user interface is designed so that the spell checking process should be as fast as possible. Instead of the ordinary processing of one misspelling at a time, many misspellings are presented at the same time, see figure 3. Since many of the reported misspellings are in fact correct words (but unknown to the program) the spell checking of the document will become much faster.

5. **Programming interface (API)**
   In order to be able to use STAVA in other applications we have constructed a simple programming interface consisting of the four simple procedures shown in figure 4.

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.

```
/* StavaReadLexicon must be called before any other function in the API. */
/* Returns 1 if the initialization succeeds and 0 otherwise. */
int StavaReadLexicon(int compound, /* 1 to allow compound words */
                     int suffix,   /* 1 to apply suffix rules */
                     int abbrev,   /* 1 to add abbreviation word list */
                     int name,     /* 1 to add name list */
                     int comp,     /* 1 to add list of computer words */
                     int correct); /* 1 to be able to correct words */

/* StavaAddWord adds a word to one of the word lists of Stava. This means
 * that in the future the word will be accepted. There are three types of
 * word lists:
 * E  - (Ending) for words that may appear alone or as last part of compound
 *      Examples: medium, fotboll, blåare
 * F  - (First) for words that may appear as first or middle part of compound
 *      Examples: medie, fotbolls, blå
 * I  - (Individual) for words that may appear only as individual words
 *      Examples: hej, du
 * Returns 1 if word could be stored and 0 otherwise. */
int StavaAddWord(unsigned char *word, /* the word to be entered */
                 char type);          /* word list type */

/* StavaWord checks if a word is correctly spelled.
 * Returns 1 if the word is correctly spelled and 0 otherwise. */
int StavaWord(const unsigned char *word);    /* word to be checked */

/* StavaCorrectWord checks if a word is correctly spelled and returns
 * ordered proposals of replacements if not. The most likely word is
 * presented first.
 * Before StavaCorrectWord is called the first time StavaReadLexicon
 * must have been called with the parameter correct=1.
 * Returns NULL if the word is correctly spelled and a string of
 * proposed replacements otherwise. If no proposed replacement is
 * found the empty string is returned. */
unsigned char *StavaCorrectWord(
              const unsigned char *word);    /* word to be corrected */
```

Figure 4: Programming interface for Stava.

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 other applications.

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 Applications

In this paper we have seen that our methods give a good spelling error detection and correction for Swedish. We have also used these methods successfully in several other applications.

8.1 Using the method on other languages than Swedish

The spelling detection and correction method described in this paper is not limited to Swedish. We have successfully used it with an English word list and some very simple suffix rules. We have also shown (but not implemented) that the method is suitable for Russian with its quite complicated inflections (Engebretsen, 1997; Axensten, 1997).

If the method is to be used on a language where inflections change letters at the beginning or middle of the word and not just at the end, the suffix rule language has to be extended, but this should be straightforward.

8.2 Spelling correction of optically scanned documents

Correction in connection with OCR is in many ways different from the ordinary spelling correction described in section 4. 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 a 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 investigations 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. a measure of distance between words, different from the one used for (directly) touch-typed texts.

8.3 Creating a part-of-speech lexicon

Building a complete part-of-speech lexicon where each word is tagged with syntactic category and inflectional morphological features is an extremely hard and time-consuming work. The work will diminish drastically when using STAVA's suffix rules extended with tagging information. All inflected forms of all regularly inflected words may be constructed automatically.

Example 7 The word *dockas* is either the genitive of the noun *docka* (doll) or the passive of the verb *docka* (dock). This is reflected by the following two suffix rules in STAVA.

$$\begin{array}{lcl} \textit{-as} & \leftarrow & \textit{-a, -an, -or} \\ \textit{-as} & \leftarrow & \textit{-a, -ade} \end{array}$$

If these rules are extended with the tags *nn.utr.sin.ind.gen* and *vb.inf.sfo, vb.prs.sfo*[2] respectively we can use the ordinary suffix rule search of STAVA to conclude that *dockas* should be tagged with the three tags *nn.utr.sin.ind.gen, vb.inf.sfo, vb.prs.sfo*.

Furthermore, by starting from the original last part list we can generate a part-of-speech lexicon. However, there are two problems with this approach: first there are no suffix rules for the inflections that are in the last part list (for example *docka*, *dockan* and *dockor*), and secondly there are no suffix rules at all for irregularly inflected words and words that are not inflected at all.

We can deal with the first problem by simply adding suffix rules also for the inflections included in the last part list. This can be done automatically by adding suffix rules for all suffixes that appear positively on the right hand side of the rule.

Example 8 For the noun suffix rules in the example above we add the following rules.

$$\begin{array}{lcll} \textit{-a} & \leftarrow & \textit{-a, -an, -or} & \textit{nn.utr.sin.ind.nom} \\ \textit{-an} & \leftarrow & \textit{-a, -an, -or} & \textit{nn.utr.sin.def.nom} \\ \textit{-or} & \leftarrow & \textit{-a, -an, -or} & \textit{nn.utr.plu.ind.nom} \end{array}$$

The second problem cannot be solved automatically. The irregular words and words without inflections have to be tagged by hand. Fortunately these are not so many in Swedish. Less than 3% (3 000 of 100 000) of the words in our last part list are of this type.

In Swedish all words in the open word classes can be inflected, which means that the number of words that have to be tagged by hand is constant.

Also note that the tagged suffix rules described above also can be used to extend an existing part-of-speech lexicon with tags for words that already are included in the lexicon. Often just the common tags for a word are included, even if uncommon tags might be necessary to know in order to be sure that the correct tagging of a word is in the lexicon.

8.4 Finding the parts of a compound word in hyphenation

In Swedish solid compounds can be very long, so there is a large need for hyphenation of compound words. The Swedish hyphenation rules say that a compound preferably should be hyphenated between the elements. This means that a Swedish hyphenation algorithm cannot only consist of local hyphenation rules. It must be able to split a compound in its elements.

We have used the method described in section 3 for doing this. In STAVA a compound is accepted if there is a way to split it into one or more elements in the *first part list* and one element in the *last part list*. If there are more than one way to split a compound every possible split is investigated and the best one is chosen. We have found that a split consisting of few elements and where the last element is long is often the correct split.

Therefore we used the following objective function for choosing between different splittings.

$$\text{Maximize } [(\text{number of characters of last element}) - 3 \cdot (\text{number of elements})]$$

[2] We have used the Swedish tagging system defined in the SUC project (Ejerhed et al., 1992).

Using the above objective function on a list of 66 000 compounds 95.5% of the compounds were split correctly, 3.0% were split incorrectly, and 1.5% were not split at all.

In order to choose splits like *kvarts-ur* (quartz watch) instead of *kvart-sur* (something like quarter sour), that is the letter *s* is moved to the first part instead of the last part in spite of that this gives a shorter last part, we changed the algorithm so that it prefers the first splitting. Unfortunately the gain was only 0.03 % (227 more words were now correctly hyphenated, but at the same time 202 words got incorrectly hyphenated).

8.5 Part-of-speech tagging of unknown words

A part-of-speech tagger typically has a lexicon consisting of words and possible taggings of these words (for example constructed automatically using the methods in section 8.3). When tagging a new text there might be unknown words, i.e. words that are not in the lexicon. The possible tags of these unknown words have to be guessed.

In Swedish the unknown words can be divided into three main groups: new compounds, proper nouns (names) and uncommon simple words (usually technical terms or dialectal words).

A compound can be split into its elements using the method in section 8.4. The tagging of a Swedish compound is decided by the tagging of its last element, so if the last element is in the lexicon we can just look up the tags.

Proper nouns can be separated from uncommon simple words in most cases since their initial letter is a capital. Otherwise we have to guess the tags from the word's appearance in some way. A good way is to use the suffix rules again. They contain both suffixes and tags, so we can look at the last few letters of the word, see if any suffix rules apply and return the corresponding tags. If we have some frequency statistics on the rules we will be able to guess which tag is the most probable.

8.6 Stemming and spelling correction in information retrieval

A common approach for an information retrieval system is to process the search question as well as the documents by removing all non-significant words (using a *stop list*) and stemming the rest of the words so that different inflections of a search term in the question and in the document does not matter.

The suffix rules in STAVA can be used for stemming. When a word matches a suffix rule we can transform it into primary form by using the first suffix on the right hand side of the rule. The problem with inflectional forms that are already in the last part list is solved by adding suffix rules as described in section 8.3.

Spelling correction can also be used in information retrieval. Up to a third of the search terms given to web search engines are misspelled. Also a large number of documents available in any given database contain misspelled terms. Since the number of untrained and novice users and low-budget text producers is increasing, the need for spelling correction in information retrieval will probably increase in the future.

The users can for example be offered interactive spelling correction of misspelled search terms. This would improve search results both as regards precision and recall. Spelling correction of the indexed documents will also improve the search results, but if this should be practically useful the correction has to be fully automatic.

We have used STAVA's spelling correction method in the web version of Skolverket's Swedish-English dictionary (Skolverket, 1997) which contains 28 500 Swedish words. Every day about 20 000 questions are asked to the web dictionary. Of these 20% are misspelled. For 33% of the misspellings a single search key is at closest distance to the misspelling, so the question can be corrected automatically.

9 Directions for future research

We have shown that the STAVA method is powerful enough to detect spelling errors and to construct and rank spelling corrections very fast. A shortcoming of the method is that it only finds spelling errors where the misspelled word is not a correct Swedish word. In many misspellings, especially of short words, the misspelled word coincides with a correct word, for example *för* (for) is easily misspelled as *frö* (seed).

A probabilistic tagger that uses word and tag frequences as well as tag bigrams and trigrams might be able to find many misspellings of this type. We will investigate this in a new project.

Especially we will look at the special case of compound splitting when for example *bokhylla* (bookshelf) is written as *bok hylla*. This type of spelling error has become more common in Swedish, probably due to English influences. By looking at the syntactic categories and frequences of both the separate words and the compound we hope to be able to find most cases of compound splitting.

In the new project we will try to detect and correct grammatical errors. When correcting a grammatical error where a word has got wrong inflectional form we know which tag the word has and which it should have. Thus we can once again use STAVA's suffix rules to construct the correction.

Having access to the tagging of the words in the document and tag frequences makes it also possible to improve the ranking of ordinary spelling corrections.

10 Acknowledgements

The research has been funded in the Language Engineering program (Språkteknologiprogrammet) by HSFR and Nutek.

We would like to thank Språkdata at the University of Gothenburg and Svenska Akademien for letting us use Svenska Akademiens ordlista as a source for words in STAVA, and Per Hedelin for letting us use the SUL word list for evaluation purposes.

References

P. Axensten. Stava ryska adjektiv (Spell Russian adjectives). Technical report, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, 1997. In Swedish. Available in WWW from
`http://www.nada.kth.se/theory/projects/swedish.html`.

B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

R. Domeij, J. Hollman, and V. Kann. Detection of spelling errors in Swedish not using a word list en clair. *J. Quantitative Linguistics*, 1:195–201, 1994.

E. Ejerhed, G. Källgren, O. Wennstedt, and M. Åström. The linguistic annotation system of the Stockholm-Umeå corpus project. Technical Report DGL-UUM-R-33, Department of General Linguistics, University of Umeå, Umeå, 1992.

L. Engebretsen. De ryska böjningsmönstrens betydelse vid maskinell rättstavning (The influence of Russian paradigms on spelling correction). Technical report, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, 1997. In Swedish. Available in WWW from `http://www.nada.kth.se/theory/projects/swedish.html`.

S. Hellberg. *The Morphology of Present-Day Swedish.* Almqvist & Wiksell, Stockholm, 1978.

R. J. Jenkins. *Dr Dobb's J.*, 22(9):107–109, 1997.

E. Johansson. Effektiv och användarvänlig svensk rättstavning under linux (Efficient and user friendly Swedish spelling correction for Linux). Technical Report TRITA-NA-E9757, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, 1997.

V. Kann. STAVA's home page, 1998. `http://www.nada.kth.se/stava/`.

K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

J. K. Mullin and D. J. Margoliash. A tale of three spelling checkers. *Software–Practice and Experience*, 20(6):625–630, 1990.

J. L. Peterson. A note on undetected typing errors. *Communications of the ACM*, 29(7):633–637, 1986.

Skolverket. *Lexin Swedish-English dictionary.* Norstedts, Stockholm, 1997. Web version available at `http://www.nada.kth.se/skolverket/swe-eng.html`.

Svenska Akademien (The Swedish Academy). *Ordlista över svenska språket (SAOL).* Norstedts Förlag, Stockholm, 11th edition, 1986.

H. Takahashi, N. Itoh, T. Amano, and A. Yamashita. A spelling correction method and its application to an OCR system. *Pattern Recognition*, 23(3/4):363–377, 1990.

M. Tillenius. Efficient generation and ranking of spelling error corrections. Technical Report TRITA-NA-E9621, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, 1996.