Implementing an efficient part-of-speech tagger[*]

Johan Carlberger   Viggo Kann
jfc@nada.kth.se   viggo@nada.kth.se
Nada, Numerical Analysis and Computing Science
Royal Institute of Technology
SE–100 44 STOCKHOLM
SWEDEN

24th March 1999

Abstract

An efficient implementation of a part-of-speech tagger for Swedish is described. The stochastic tagger uses a well-established Markov model of the language. The tagger tags 92% of unknown words correctly and up to 97% of all words. Several implementation and optimization considerations are discussed.

The main contribution of this paper is the thorough description of the tagging algorithm and the addition of a number of improvements. The paper contains enough detail for the reader to construct a tagger for his own language.

Keywords: part-of-speech tagging, word tagging, optimization, hidden Markov models.

Introduction

In part-of-speech (POS) tagging of a text, each word and punctuation mark in the text is assigned its morphosyntactic tag. Different tagging systems use different sets of tags, but typically a tag describes a word class and some word class specific features, such as number and gender. The number of different tags varies between a dozen and several hundred.

Constructing an automatic part-of-speech tagger involves two problems:

1. to find the possible tags for each word. This is easy to do if the word is in a word-tag lexicon, but if the word is unknown, the possible tags have to be guessed.

2. to choose between the possible tags. This is called *syntactic disambiguation*, and it has to be solved for each word that is ambiguous in its part-of-speech. Ambiguous words are very common in most languages, for example the English word *set* can be either a noun, an adjective, or a verb.

The level of analysis needed for correct disambiguation varies from sentence to sentence. To correctly disambiguate the word *turn*, which is a noun in *make a turn here*, but a verb in *I will turn off the E4*, requires only syntactic analyses of the sentences. But

[*]To appear in Software Practice and Experience, 1999.

whether *off* in the second sentence is a particle or a preposition depends on whether E4 is the name of a road or a device. Thus, 100% correct automatic tagging is a rather impossible goal, since it requires full understanding of the text. Tagging accuracy in different systems described in the literature varies between 95% and 98%.

There are several important language technology applications that may include part-of-speech tagging as a first step, for example machine translation, information retrieval, and grammar checking. The applications require the tagger to be both efficient (to tag fast, especially important in information retrieval), and accurate (to tag correctly, especially important in translation). In some applications it is not even enough to have the text syntactically disambiguated – a *word sense disambiguation* is needed, and that is an even harder problem [1].

Part-of-speech taggers can be constructed in various ways, and different types of taggers have different advantages. Taggers can be based on stochastic models (for example [2, 3, 4, 5, 6, 7]), on rules ([8, 9]), or on neural networks ([10]) In a recent paper, Samuelsson and Voutilainen claim that rule-based taggers can give higher tagging accuracy than plain stochastic taggers on correct texts [11]. But hybrids between rule-based taggers and stochastic taggers might be even better [12].

Some different stochastic models for tagging unknown words exist ([2, 4]). A good survey of automatic stochastic part-of-speech tagging is [13].

In this paper we describe an implementation of a part-of-speech tagger for Swedish. We wanted the tagger to be easy to implement, fast, language independent, tag set independent, and that it should give high accuracy of tagging. We also wanted the tagger to be able to cope with unknown words and grammatically erroneous sentences. This ability is needed in various applications, such as grammar and spell checking.

Given these requirements, we chose to construct a stochastic tagger based on a Markov model. Our goal was to achieve 95% tagging accuracy for known words and 70% accuracy for unknown words, and we both reached and surpassed the goal.

We use the tagger in a grammar checking program for Swedish, named GRANSKA, but we designed it to be as language independent as possible, and we think that it can be used for most inflectional languages, for any tag set, and in any application needing part-of-speech tagging. As it turned out, when incorporated into GRANSKA, our tagger actually became a hybrid between a stochastic tagger and a rule-based tagger. For certain complicated cases where the stochastic tagger could be wrong we use rules to find the correct tagging.

The Tagging Model

Markov model

In this section we briefly describe the Markov model that is used as a stochastic model of the language. A complete and excellent description of the equations used in the standard Markov model for part-of-speech tagging is found in [2].

A text of n words is seen as a sequence of random variables $W_{1..n} = W_1 W_2 \ldots W_n$, and the corresponding tagging is also a sequence of random variables $T_{1..n} = T_1 T_2 \ldots T_n$. A particular sequence of values of $W_{1..n}$ ($T_{1..n}$) is denoted $w_{1..n}$ ($t_{1..n}$). The definition of the tagging problem is then

$$\mathcal{T}(w_{1..n}) = \arg\max_{t_{1..n}} P(t_{1..n}|w_{1..n}),$$

where the operator arg max computes the tagging maximizing the probability, according to the model, that word sequence $w_{1..n}$ is tagged $t_{1..n}$.

A second order Markov model makes the two assumptions

$$\begin{aligned} P(w_i|t_{1..i}, w_{1..i-1}) &= P(w_i|t_i), \\ P(t_i|t_{1..i-1}, w_{1..i-1}) &= P(t_i|t_{i-2}, t_{i-1}), \end{aligned}$$

that is, the word itself only depends on its tag, and the tag only depends on the two preceding tags in the text.

The tagging problem can now be formulated as

$$\mathcal{T}(w_{1..n}) = \arg\max_{t_{1..n}} \prod_{i=1}^{n} P(t_i|t_{i-2}, t_{i-1}) P(w_i|t_i). \tag{1}$$

An unattractive feature of this formulation is that the quantities $P(w_i|t_i)$ are very small and difficult to estimate. Since the reversed conditional probabilities $P(t_i|w_i)$ are much more attractive in this respect, the following is a plausible alternative:

$$\mathcal{T}(w_{1..n}) = \arg\max_{t_{1..n}} \prod_{i=1}^{n} P(t_i|t_{i-2}, t_{i-1}) P(t_i|w_i). \tag{2}$$

Both these equations (and in particular, their corresponding first order Markov model equations) have been used in different stochastic taggers, but in [2], the two equations were compared, and Equation 1 was found to be significantly better when tagging texts with quite a large training text[1].

If all the probabilities are known, the optimal solution to the tagging problem using Equation 1 is most efficiently computed with dynamic programming using the so called *Viterbi algorithm* [14]. This algorithm avoids the polynomial expansion of a breadth first search by trimming the search tree at each level; see our implementation in Figure 2. The time complexity of the algorithm is linear in the number of words to be tagged.

Type of statistics used

In the rest of this paper we mean by a *word token* a specific occurrence of a word in a text, as opposed to a *word type*, which means a lexically unique word. If no misunderstanding is possible, we sometimes write just *word*.

Before the Viterbi algorithm can be used we have to give estimates of the probabilities used in the Markov model. We obtain these estimates by collecting statistics of a large *tagged corpus*, which is a large text (typically consisting of a million words or more) where each word token is tagged with its supposedly correct morphosyntactic tag. From such a corpus we collect the statistics in Table 1 and compute estimates for the probabilities as described in Table 2.

Designing a Tagging Algorithm

Modifying the Markov model equations

A problem with the probability distributions of Equations 1 and 2 is that the training text is often too small to give accurate estimates (the sparse data problem). In this

[1]No theoretical justification for using (2) is known [2]. Note, however, that (2) is equivalent to including the factor $P(t_i)$ in (1). Therefore, (1) gives preference to common tags in some cases where unusual tags are more probable. In this sense (2) is more conservative.

Table 1: *Statistics to be collected.*

notation	**counting the number of**
C_n	all word tokens w
$C(w)$	occurrences of the word w
$C(w,t)$	occurrences of the word w tagged with t
$C(t)$	occurrences of the tag t
$C(t_1,t_2)$	occurrences of the *tag bigram* (t_1,t_2), that is the tag t_1 followed by the tag t_2
$C(t_1,t_2,t_3)$	occurrences of the *tag trigram* (t_1,t_2,t_3), that is the tag t_1 followed by t_2 followed by t_3
$C(w_1,t_1,t_2)$	occurrences of the *wordtag-tag bigram* (w_1,t_1,t_2), that is the word w_1 tagged with t_1 followed by the tag t_2
$C_m(t)$	different word types tagged with tag t
$C_c(t)$	occurrences of capitalized words tagged with t
$C_m(w_{\text{end-i}},t)$	different word types ending with the same i letters w and tagged with t

Table 2: *Estimates of the probabilities.*

$P(t_i) = \frac{C(t_i)}{C_n}$	$P(t_i \mid w_{i-1},t_{i-1}) = \frac{C(w_{i-1},t_{i-1},t_i)}{C(w_{i-1},t_{i-1})}$
$P(t_i \mid t_{i-1}) = \frac{C(t_{i-1},t_i)}{C(t_{i-1})}$	$P(w_i \mid t_i) = \frac{C(w_i,t_i)}{C(t_i)}$
$P(t_i \mid t_{i-2},t_{i-1}) = \frac{C(t_{i-2},t_{i-1},t_i)}{C(t_{i-1},t_{i-2})}$	$P(t_i \mid w_i) = \frac{C(w_i,t_i)}{C(w_i)}$

section, we investigate some previously reported ways of smoothing and modifying the probabilities and devise an equation for our tagger.

As reported in [2] Equation 1 gives significantly better results than Equation 2, in our case 96.3% compared to 95.2%. A heuristic attempt to interpolate Equation 1 and 2 was investigated. This interpolation was accomplished at no extra cost in terms of efficiency, since the lexical probability ($P(w_i|t_i)$, or corresponding variant) for each word is computed just once when constructing the lexicon the first time, as described in the section Implementation Considerations. But as there was no significant improvement, we decided to base our model just on Equation 1.

Åström claims [15] that the following modification to the lexical probability did enhance the tagging accuracy for another stochastic tagger using the same corpus:

$$P_{\text{new}}(w|t) = P(w|t)^k,$$

where the optimum value for the parameter k was 0.6. The effect of this modification is that the probabilities of common word-tags are reduced, thereby making it easier for the tagger to choose lexically less probable tags. Tests with our tagger gave an optimum for k very close to 1 and with no significant improvement, so this modification was discarded.

The tag trigram probability estimate $C(t_{i-2},t_{i-1},t_i)/C(t_{i-2},t_{i-1})$ presents a problem, since, unless a very large training text is used, the tag trigram and bigram counts will be low, as was the case for our corpus (see Implementation Considerations). The common way to improve the model is to interpolate between the tag trigram, bigram, and unigram probabilities. Hence, we redefine the trigram probability estimate as follows:

$$P_{int}(t_i|t_{i-2},t_{i-1}) = \lambda_1 \tfrac{C(t_{i-2},t_{i-1},t_i)}{C(t_{i-2},t_{i-1})} + \lambda_2 \tfrac{C(t_{i-1},t_i)}{C(t_{i-1})} + \lambda_3 \tfrac{C(t_{i-1})}{Cn}, \tag{3}$$

where the optimization parameters satisfy $\lambda_1 + \lambda_2 + \lambda_3 = 1$. This modification improves the model, but the tag trigram probabilities are still unreliable when the bigram counts $C(t_{i-2},t_{i-1})$ are small. This effect can be reduced by making λ_1 a function of the number of such bigrams, so that the trigram contribution is stronger when the bigram count is high. Experiments showed that

$$\lambda_1 = \lambda_0 C(t_{i-2},t_{i-1})^{\alpha},$$

where the optimization parameter α is about 0.2, gave a slight improvement.

Another problem with Equation 3 is that the trigram count and the two bigram counts in it can be zero. We experimented with using small numbers ε instead of zero to smooth the equation. It turned out that using a single ε instead of the trigram term when it was zero or 0/0 worked fine:

$$P(t_i|t_{i-2},t_{i-1}) = \max\left(\lambda_0 \frac{C(t_{i-2},t_{i-1},t_i)}{C(t_{i-2},t_{i-1})^{1-\alpha}}, \varepsilon\right).$$

Another improvement of Equation 1 could be to condition the next tag on the previous word, as well as on the two previous tags. As pointed out in [13], such an extension may not be a good idea, since the wordtag-tag bigrams will most likely be too sparse. A manual inspection of the tagging errors, however, suggested that including wordtag-tag bigrams could indeed enhance the performance.

Not surprisingly, no improvement of tagging accuracy was obtained when the following wordtag-tag bigrams term was added to Equation 1:

$$P(t_i|w_{i-1},t_{i-1}) = \frac{C(w_{i-1},t_{i-1},t_i)}{C(w_{i-1},t_{i-1})}.$$

In an effort to reduce the bad influence of sparse data, the extra term was used only for wordtag-tag-bigrams with a frequency higher than a certain limit. However, there was still no improvement. We resorted to inspect the impact of each word type occurring as w_{i-1} on the performance when used in the wordtag-tag bigram formula. This experiment revealed that many word types worsened the performance, but a few word types actually improved performance. Thus, we modified Equation 1 to condition on the previous word (as well as the two previous tags) only when the previous word has proved to improve performance on the optimization text:

$$P(w_{1..i-1},t_{1..i-1}) = \begin{cases} P(t_i|w_{i-1},t_{i-1},t_{i-2}) & \text{if } w_{i-1} \text{ is a 'good' word,} \\ P(t_i|t_{i-2},t_{i-1}) & \text{otherwise,} \end{cases}$$

where words that improved the model are termed 'good'. The introduction of the wordtag-tag bigrams modification of Equation 1 gave a 7% improvement of tagging accuracy on the optimization text, but no significant improvement on the test text. A much larger corpus than the one we use is required to be confident about the improvement of the model, since many of the good words occur very few times in the test and optimization texts.

Despite the failure of introducing word-bigrams, we still believe that it is possible to select wordtag-tag bigrams that do improve the model, for example verb-particles and verb-prepositions. We have not yet made such an attempt.

Using capitalization to improve tagging

In Swedish, all proper nouns should be spelled with an initial capital, and all other words (not commencing a sentence) should be spelled with a noncapital initial. Thus, the model could be modified to only select proper noun tags for capitalized words, but since the capitalization convention is often not fully observed (as was the case in our corpus), we introduced the following factor to Equation 1:

$$P_c(w,t) = \begin{cases} \gamma_1 & \text{if } t \text{ is not proper-noun tag and } w \text{ is capitalized,} \\ \gamma_2 & \text{if } t \text{ is proper-noun tag and } w \text{ is not capitalized,} \\ 1 & \text{otherwise.} \end{cases}$$

The best performance was obtained with $\gamma_1 = 0.028$ and $\gamma_2 = 0.044$. (In the case for unknown words, $\gamma_1 = 0.020$ and $\gamma_2 = 0.048$.)

The modification has the effect that an initial capital does not necessarily lead to the word being tagged with a proper noun tag, and noncapitalized words may be tagged as proper nouns. Tagging accuracy improved from 96.2% to 96.3% by using P_c.

Thus, we use the following equation in our language model:

$$\mathcal{T}(w_{1..n}) = \arg\max_{t_{1..n}} \prod_{i=1}^{n} P_{int}(t_i|t_{i-2},t_{i-1})P(w_i|t_i)P_c(w_i,t_i), \text{ where} \tag{4}$$

$$P_{int}(t_i|t_{i-2},t_{i-1}) = \max\left(\lambda_1' \frac{C(t_{i-2},t_{i-1},t_i)}{C(t_{i-2},t_{i-1})^{1-\alpha}}, \varepsilon\right) + \lambda_2 \frac{C(t_{i-1},t_i)}{C(t_{i-1})} + \lambda_3 \frac{C(t_{i-1})}{Cn}.$$

Markov model parameter optimization

The standard way to train a hidden Markov model is by maximum likelihood training using the forward-backward algorithm, also known as the Baum-Welch algorithm [13]. This algorithm uses word type, rather than tag information, to iteratively improve the probabilities of the training corpus. The advantage of this approach is that it does not require the training corpus to be tagged.

If there is a tagged training corpus available, the probabilities in the Markov model can be estimated using relative frequencies in the tagged text. This method is more reliable, so we used it in our tagger. If there is an untagged corpus beside the tagged corpus, one could try to improve the probabilities computed using relative frequencies by a maximum likelihood training. We did not do this, because Merialdo has shown that no improvement will occur if the tagged corpus is large enough [7].

As the tagging of unknown words described in the following section and the heuristic modifications that we have introduced add several parameters to the original model, standard optimization algorithms, such as the forward-backward algorithm, become harder to use. Instead, we use a variant of simulated annealing. The initial parameter values are set arbitrarily, and the optimum values, which give the highest tagging accuracy of the optimization text, are determined by simple linear searches. Then the values are slightly altered from the optimum, and the optimization is repeated. This operation avoids the parameters getting stuck in local maxima. Although naive, the method turned out to be quite practical, especially when new parameters are introduced and evaluated during development. Also, the method does not put any restrictions on the equation to be optimized.

Less naive than a linear search for the optimum parameter values would be to use golden section search [16]. Such an algorithm was implemented, but the optimization text proved too small to give high enough variation in tagging accuracy for the algorithm to work properly.

Implementation of the Markov model

Before a text is tagged it must first be tokenized, i.e. each word and punctuation mark must be identified. Tokenization is a not a trivial task, mainly because periods are used in abbreviations and numbers, as well as full stops [17].

Using Flex we have implemented a simple tokenizer that is used by the tagger. As the text is tagged and checked with error rules, the text may have to be re-tokenized. As our corpus was already tokenized, the evaluated performance of the tagger is not degraded by poor tokenization.

During tokenization, the text is divided into sentences, and then each sentence is tagged separately. It makes little sense to cross sentence boundaries, as the correlation between words and tags in adjacent sentences is often very weak. Also, as the scope of the algorithm is only three words, very little information traverses sentence boundaries anyway.

In the dynamic programming implementation of the Viterbi algorithm we need a data structure, an object named `TrigramGadget`, for each position in the sentence to be tagged. A `TrigramGadget` keeps track of the possible tags in a specific position, the probabilities calculated so far for all combinations of tags of the two last positions, and the best tag in the position before the last two.

When tagging a new sentence we first initialize an array `gadget` of `TrigramGadget`s by setting the first two tags to a sentence-delimiter tag, as seen in Figure 1. The sentence-delimiter tag is introduced to make the algorithm work properly. In the training corpus two such tags are inserted between every sentence in order to get proper statistics.

```
TrigramGadget gadget[MAX_SENTENCE_LENGTH];
gadget[0].SetTags(sentenceDelimiterPseudoWord);
gadget[1].SetTags(sentenceDelimiterPseudoWord);
```

Figure 1: *Initialization of the gadget used in the dynamic programming.*

After initialization we walk through each word in the sentence and compute the probability according to Equation 4 for each possible combination of tags for the trigram ending with the current word. The best probability for each possible tagging of the last two words is then stored in `gadget`. When the end of the sentence is reached, two sentence-delimiter tags are inserted, and then the algorithm works backwards through all positions in the gadget to retrieve the best tagging sequence. The whole algorithm is found in Figure 2.

Tagging Unknown Words

A simple approach

When the tagger encounters an unknown word in the text, obviously Equation 4 cannot be used straight away, since there is no statistical information gathered for that particular word. But if we could in some way estimate a lexical probability $P_u(w|t)$ for unknown words, the lexical probabilities $P(w|t)$ could be substituted by $P_u(w|t)$ in Equation 3 when w is unknown.

A simple approach to estimate $P_u(w|t)$ would be to count the number of word types

```
TagSentence(Sentence *s) {
  Word *w1 = period;
  for (int i=0; i<=s->NWords(); i++) {
    TrigramGadget &g2 = gadget[i+2];
    TrigramGadget &g1 = gadget[i+1];
    TrigramGadget &g0 = gadget[i];
    WordToken &t = s->wordTokens[i];
    // t contains the current word token and subsequently the chosen tag
    Word *w2 = t.word;
    SetLexicalProbs(w2, w1, *t, g2);
    // assigns the possible tags and lexical probabilities of w2 to g2.
    for (int u=0; u<g2.n; u++) {
      const Tag *tag2 = g2.tag[u];
      for (int v=0; v<g1.n; v++) {
        const Tag *tag1 = g1.tag[v];
        probType best = 0;
        for (int z=0; z<g0.n; z++) {
          const Tag *tag0 = g0.tag[z];
          probType prob = g1.prob[v][z]*tags.Pt1t2t3(tag0,tag1,tag2);
          if (prob > best) {
            best = prob;
            g2.prev[u][v] = z; // index in position 0 giving best probability
          }
        }
        g2.prob[u][v] = best*tag2->lexProb;
      }
    }
    w1 = w2;
  }
  // finally walk backwards through gadgets and extract best tag sequence:
  Rewind(s, gadget+2);
}
```

Figure 2: *Heart of the tagging algorithm.*

$C_m(t)$ that have been tagged with tag t in the training text, and to use

$$P_m(w|t) = \frac{C_m(t)}{\sum_{\tau \in \text{tag set}} C_m(\tau)}$$

as an estimate. Using this estimate of $P_u(w|t)$ results in 45.5% of the unknown words being tagged correctly. We can do much better, however, by performing a statistical morphological analysis of unknown words. Intuitively, by just looking at the last few letters of a word, a fair guess of the syntactic functionality of that word can be made. This is true for Swedish, English, and other inflectional languages.

Analyzing word-endings

Ideally, a morphological analysis of unknown words should be done by a program that finds roots and suffixes of these words, for example as described in [2]. Devoid of such a program, we examined a method that does not require any additional information about the morphology of words than what we already have. We simply count the number of word types with common endings of length i, $C(w_{\text{end-i}}, t)$, for each tag t in the tag set. The endings used are just all occurring word-endings of length ranging from 0 to L. Then an estimate is made by

$$P_e(w|t) = \sum_{i=0}^{L} \alpha_i \cdot \frac{C(w_{\text{end-i}}, t)}{\sum_{\tau \in \text{tag set}} C(w_{\text{end-i}}, \tau)},$$

where α_i are optimization parameters. Tagging accuracy increased with increasing L, but no significant improvement was detected for L greater than 5. By using this estimate, an encouraging 88.7% of the unknown words were correctly tagged. P_m is actually P_e with L=0, so we use P_e instead of P_m.

Analyzing compound words

Contrary to the English language, a large portion of Swedish words are compounds. As compounds are frequent among unknown words, they deserve special attention. If a successful decomposition of an unknown compound word can be made, we have strong evidence of the correct possible tags for that word, since the last word form in a compound determines its part-of-speech. In STAVA [18, 19, 20] an algorithm for decomposing compounds into their word form parts was implemented.

Awaiting the incorporation of STAVA into the program, we use a simplified algorithm that works in the following way: A possible compound w is divided into two parts with an optional letter "s" in between: $w = w_p$(s?)w_s. If a suffix w_s of a word w happens to be another word in the lexicon we use the lexical probabilities of w_s to improve on the estimate of P_u for w. If the first part w_p of the possible compound is also a word in the lexicon, it is more likely that w is a compound than if the first part is not a word in the lexicon. Contrary to the STAVA compound decomposer, this method does not verify that the compound is correct, and it does not consider all possible ways of compound construction. Thus, P_s was defined as:

$$P_s(w|t) = \begin{cases} \alpha_s P(w_s|t) & \text{if } w_s \text{ is a suffix of } w, \\ 0 & \text{otherwise.} \end{cases}$$

Introducing P_s improved tagging accuracy of unknown words to 92.0%. Initially, the compounds were categorized whether a prefix word was found or not, and using different weights α_s to reflect the assumption that a found prefix indicates a correctly identified compound, but as there were very few incorrect analyses, this categorization did not improve performance.

Resulting lexical probability

The final P_u was then defined as

$$P_u(w|t) = P_e(w|t) + P_s(w|t).$$

In this approach to tag unknown words, both morphology and context are taken into account. An experiment using an Åström exponent (see Modifying the Markov model equations) on P_u in order to monitor the relative impact of morphology versus context was carried out, but no improvement in performance was reached.

As a further optimization, the set of tags considered could be reduced, since about half of the tags in the tag set are tags for function words (prepositions, determiners, etc.). These classes of words can be considered closed, as the lexicon should contain all such words. Consequently, only tags for content words (nouns, verbs, etc.) need to be considered.

When P_u has been computed for all content tags, only the best N tags are used in Equation 4. When N was varied during optimization, tagging accuracy increased with increasing N, but no significant improvement was detected for N greater than 5. This is good because Equation 4 takes time proportional to N^3 for each word to compute.

Implementation Considerations

In this section we discuss some problems of the corpus we used and some modifications to it. We also describe a number of implementation details concerning speed and memory usage. For efficiency and portability reasons we chose to implement the program in C++.

Selecting a corpus

The only currently available tagged Swedish corpus is the Stockholm-Umeå Corpus (SUC) [21], which made the choice of corpus easy. SUC consists of one million words divided into 500 texts of 2 000 words each. The tag set consists of 140 different tags. As could be easily observed, the corpus was too small to give reliable tag trigram counts for our model. The average count of occurring tag trigrams is merely 12, and no less than 41% of the occurring tag trigrams occur just once. Furthermore, many tags in the tag set have very few occurrences in the corpus (34 tags occur less than 100 times, and 17 tags occur less than 10 times), which clearly makes statistical observations unreliable.

The corpus was divided into a training part, an optimization part, and an evaluation part. The optimization and evaluation parts consist of 14 000 words each.

Statistics were extracted from the corpus by using the UNIX utilities flex, tr, sort, uniq, cut, wc, comm, and sed.

Modifying the tag set

To make the SUC tag set better suit our purposes, we decided to remap the tag set in two ways. Firstly, we removed some of the least common tags by uniting two or more tags into one single tag, thereby removing the most unreliable data. Also, reducing the number of tags reduced the amount of information to be retrieved and stored in the lexicon. Secondly, we introduced new tags when we thought the original classification to be too coarse.

For example, in SUC there is no distinction between auxiliary verbs (*be*, *have*, etc.) and other verbs (*run*, *eat*, etc.). These two types of verbs clearly have different syntactic behavior, which motivates the introduction of a new tag for auxiliary verbs. Also, SUC uses the same tag for cardinals in singular (*1*, *en* (one), etc.) and plural (*2*, *två* (two), etc). Therefore, we introduced a number feature to the cardinal tag.

We restricted ourselves to only extend the tag set when the remapping of the corpus can be made automatically. A remapping requiring manual inspection would mean too much work.

The task of evaluating the impact of the modifications of the tags is rather time consuming, since it involves building a new lexicon and optimizing the tagger. Therefore, we confined ourselves to evaluate the impact of the total remapping of the tag set, leaving us unaware of the quality of each single modification. The evaluation showed that removing 14 tags and adding 5 new tags improved tagging accuracy by a modest 2%.

Adding extra words to the lexicon

No matter how large the training corpus, there will be a considerable amount of unknown words in the texts to be tagged. Fortunately, we had access to Svenska Akademiens Ordlista (SAOL), a word list containing about 100 000 noncompound words, in most cases word class classified [22]. By using and modifying an existing word form generator [20], all possible word forms were generated from each word base form in SAOL. Then all generated word forms including the base form were assigned all possible SUC tags, in order for these words to be included in the existing SUC lexicon. This extension increased the number of known word-tags from 104 000 to 673 000.

There was one problem, however. We had no statistical information available to estimate the lexical probabilities of the SAOL-generated words, so by some other means we had to estimate these probabilities. The same problem concerned unknown words encountered during tagging of a text. The simple solution we chose was to make use of the morphological analysis of unknown words described in section Tagging Unknown Words. This analysis gives a probability distribution $P_e(w|t)$ over the tag set based on the morphology of the word. As these probabilities in some way reflect how common different word forms are, it makes sense to make use of this estimate here. These probabilities can be used straight away for SAOL words that were unknown to SUC. However, SAOL word-tags consisting of a known SUC word, but an unknown tag for that word, present a problem.

How can the existing lexical probabilities be extended with extra probabilities for new tags in a good way? If the word count is high the existing lexical probabilities are probably good, but not if the word count is low. The following heuristics showed to increase the tagging accuracy for such words:

$$P_{\text{SUC+SAOL}}(w|t) = \begin{cases} P_e(w|t) & \text{if } (w,t) \text{ only in SAOL,} \\ \delta_1 P_e(w|t)/C(w)^{\delta_2} & \text{if } w \text{ in SUC, } (w,t) \text{ only in SAOL,} \\ P(w|t) & \text{if } (w,t) \text{ in SUC,} \end{cases}$$

where $\delta_1 \approx 0.00008$ and $\delta_2 \approx 0.02$ are optimization parameters. The effect is that a SAOL tagging of a word has greater influence for less common SUC words.

Adding all SAOL words and word-tags reduced the number of unknown word tokens in the test text from 6.6% to 5.2% and reduced the number of unknown word-tags from from 0.53% to 0.30%. The tagging accuracy increased from 96.1% to 96.3%. As the tagging accuracy of unknown words compared to that of known words is quite good, no dramatic improvement of tagging accuracy can be expected by lexicon expansion.

A more economic way to use the SAOL word forms was to add only SAOL word-tags for known SUC words. This approach, in combination with generating word-tags for known SUC words that were not derived from SAOL, reduced the number of words that had never been tagged with the correct tag in the training text from 0.53% to 0.15%, and tagging accuracy increased from 96.1% to 96.3%.

Fast loading of lexicons

One of the design requirements for the tagger was that it should be fast, which also means that the program must load its lexicon fast. The ideal situation would be that the lexicon structures have no pointers or other references to memory locations. Then the lexicon can be loaded in big chunks of data from files directly into memory. Avoiding pointers completely, however, would be too impeding on the quality of the code, but by avoiding dynamic structures when appropriate, very fast loading is possible.

Given the tagging model we use, the data collected from the training text can be considered static, so there is no need for dynamic structures for the main lexicon. The information about each word in the main lexicon is of equal size except the string of the word and the number of different tags associated with the word. Allocating memory for all word-strings and all word-tags at the same time is the obvious way to save space and time.

Furthermore, word-strings like *pådrag* and *drag* share the last four letters and can therefore be stored in the same location. The sharing of memory locations reduced the total size of all word strings by 11%. Since the same lexicon structure is used for storing statistics about word-endings, as described in section Tagging Unknown Words, the total size of all word-ending strings was reduced by 31%.

When the main lexicon is stored to a file, the word-strings and word-tags are stored in big chunks of data. The actual memory location pointer is also stored. When the word-strings and word-tags are loaded again, they probably end up in a different memory location. In this case, all references from word-structures to word-strings and word-tags are changed appropriately. This method effectively combines use of pointers and high-speed storing and loading, but of course requires careful implementation to avoid erroneous pointers.

Initially the main lexicon data is stored in a number of files created during the extraction of statistics from the training corpus. When the lexicon is loaded the first time, hash tables and other structures are constructed, and then the lexicon is stored to a file in a “fast” format. This format is basically the memory contents and some information of sizes and memory locations. The next time the lexicon is loaded, the program automatically uses the fast format files. The slow and fast loading modes allow the original lexicon files to be in a convenient format, since there is no requirement for speed during the initial slow loading. All subsequent loadings of the same lexicon will, on the other hand, be very fast.

Further speed optimizations

The main lexicon words are stored in a static hash table, where the slots in the table are the word-structures themselves, not pointers to word-structures. This design saves one indirection for every look-up, and it saves one pointer for each word. Moreover, the size of the table is the same as the number of words, which means no memory loss. Collisions in the hash table are resolved by using one link per slot, where the link is an index to another place in the table. This design does not use any pointers, and hence the hash table itself can be responsible for fast storing and loading, provided that the hash table objects themselves can be relocated in memory.

Hash tables require a hash function on its objects, and the simple hash function in Figure 3 on the word-strings turned out to be satisfactory. Using this hash function, the average number of collisions in the table is only 2.2, and the maximum number of collisions is 7.

The static hash table is used for the main lexicon words, wordtag-tag bigrams, word-endings, tags, and tag trigrams. As unknown words are detected during tagging they are stored in a standard dynamic self-resizing hash table.

```
int key(string *s)
{ int val = 0;
  for (s; *s; s++)
    val = (val >> 1) xor scatter[(uchar) *s];
  return val;
}
```

Figure 3: *Hash function used in the program. The array* `scatter` *is a function from [0..255] to a random number [0..maxint].*

We experimented with storing the key of each hash table object in the object to avoid multiple computations of the hash function, both for words and for tag trigrams, but it did not increase tagging speed.

In order to optimize speed, all optimization parameters are declared as constants in the release-version of the program. Also, pre-computing mathematical expressions where possible, for example the trigram probabilities and the lexical probabilities in Equation 4, speeds up tagging.

One problem with Equation 4 is that when long sentences are tagged, the overall probability will approach zero. This problem can be avoided by using log-probabilities and transforming the equation to a sum of terms, or by normalizing the probabilities half-way through a sentence when the probabilities grow too small. We chose the latter approach, since, when using 4 bytes for storing the probabilities, only very few sentences required normalization, and none when 8 bytes were used. Also, using log-probabilities would require the equations for unknown words to be transformed too, which is not easily done. Alternatively, it would require the resulting probability distribution $P_u(w|t)$ to be logarithmized at run-time, which is quite time-consuming.

Applications of the Tagger

A probabilistic tagger can be used in many different applications. We discuss two applications: grammar checking and spell checking.

Grammar checking

We constructed the tagger in order to apply it in a Swedish grammar checking program called GRANSKA. As we conclude in the following section, the syntactic disambiguation dramatically improved the quality of the grammar checking, with respect to both the possibility of finding errors and to the false alarm rate.

The grammar checking algorithm discovers grammatical errors using *error rules*. Each rule describes an erroneous construction in a notation that uses the part-of-speech tagging of the text. Figure 4 shows an example of an error rule.

In such an application, the demands on the tagger are somewhat different than in other applications.

Firstly, an incorrectly tagged word is not a problem if it occurs in a correct sentence, unless the incorrect tagging triggers an error rule, but that is unlikely to happen. Secondly, the tagger has to be able to tag text containing errors in some reasonable and consistent way, such that it is possible to express the different types of errors in error rules. We conjecture that, in general, stochastic taggers tag texts with errors better than rule-based taggers.

```
{
  x/wordcl=dt & spec=def/,                         % a definite article
  y/wordcl=jj & gen=x.gen & num=x.num & spec=ind/, % an indefinite adjective
  z/wordcl=nn & gen=x.gen & num=x.num & spec=def/  % a definite noun
  -->////"The adjective " y " does not agree with the noun " z//
}
```

Figure 4: *An error rule in* GRANSKA *discovering an incongruence between an indefinite adjective and a definite noun.*

Interestingly, it is also possible to use the error rule matching to improve the tagging. In cases where the stochastic tagger makes errors in a systematic way (for example often tags the word *x* as *y* instead of *z*), we can write an error rule that recognizes the erroneous tagging, re-tags the sentence forcing the tagger to tag *x* as *z*, and then re-applies the error rules.

Such a correction rule could, for instance, be used to capture simple feature agreements that cannot be captured by the Markov model, due to its short scope. For example, our second order Markov model cannot decide whether the noun *bord* is in singular or plural in the sentence *Några mycket bra bord* (Some very good tables), because the number of the last word is only revealed by the number of the first word. In such cases it is a simple matter to construct a correction rule to help the tagger select the correct tag.

Since this type of *correction rules* is a sort of rewriting rule similar to the ones used by rule-based tagging algorithms, our tagger can be called a hybrid tagger using both stochastic and rule based methods, something that is recommended as the best technique to construct high-quality taggers [12]. An interesting difference between the hybrid tagging approaches is that the tagger in [12] first uses rule based methods and then stochastic methods, while our tagger first uses stochastic methods, then rule based methods, and then perhaps new rounds of stochastic and rule based methods.

Unfortunately, we have not yet been able to evaluate the tagging obtained using correction rules, but we think it would be a substantial improvement.

Probabilistic spell checking

With a working probabilistic tagger at hand, we have started to develop a probabilistic spell checker. The spell checker identifies a suspicious sentence (using some measure related to the probability obtained with Equation 4), makes changes to the sentence in order to make it less suspicious, and if a good enough alternative is found, suggests it as an alternative to the original suspicious sentence. This method captures not only errors that “traditional” spell checkers findm, but also errors where the misspelled word happens to be a word in them word list, or when adjacent are words interchanged, or when a is omitted, or when a an extra word has slipped into the sentence by mistake.

If this application can be made successful enough, it could correct some types of text errors normally taken care of by grammar checkers, and could thus be incorporated into a rule based grammar checker. The spell checker we constructed successfully identified and corrected the following incorrect versions of the sentence *Jag har en bil* (I have a car):

Table 3: *The tagging results.*

sentences	1006
words (including punctuation marks)	16378
unknown words	1085 (6.6%)
hard words	25 (0.15%)
correctly tagged sentences	602 (58.6%)
correctly tagged words	15775 (96.3%)
unknown word errors	87 (14.4% of errors)
hard word errors	25 (3.9% of errors)
known word-tag errors	510 (84.6% of errors)
accuracy for unknown words	92.0%
accuracy for known words	96.6%
accuracy for all words	96.3%

Jag har en bik.	(unknown word *bik*)
Jag har en bi.	(noun *bi* causes incongruence)
Jag en har bil.	(swapped words)
Jag har en har bil.	(superfluous word *har*)
Jag en bil.	(missing verb)

Whether the spell checker can successfully correct more complicated sentences remains to be investigated, but as the toy examples show, the method seems promising.

Evaluation of Performance

Speed and memory requirements

A main lexicon constructed from the statistics derived from SUC only uses 7 MB of memory. A SAOL-extended SUC lexicon occupies 29 MB.

Using the SUC lexicon (containing 104 000 word-tag pairs), the tagger loads in 0.79 seconds and tags 26 900 words per second on a Sun Sparcstation Ultra 1. Using both the SUC and SAOL lexicons (674 000 word-tag pairs), the tagger loads in 1.9 seconds and tags 22 600 words per second. Thus, there is almost no speed degradation for tagging with very large lexicons.

Quality of tagging

SUC consists of one million words divided into 500 texts of 2 000 words each. We have separated two randomly selected 14 000 word texts from this corpus, one for optimization and the other for evaluation.

The tagging results are quite encouraging. Using Equation 4, we obtained a tagging accuracy of 96.3%. In this text, 6.6% of the word tokens were unknown to the tagger, and 92.0% of these words were tagged correctly, which is far better than our initial goal of 70%. The results of the test text tagging are presented in Table 3.

An analysis of the incorrectly tagged words revealed that 14.4% were unknown words, and the tagger made a bad choice. 3.9% were “hard” words, i.e. known words that had never been tagged with the wanted tag in the training text. These words are named hard because the current model cannot tag them correctly. In the remaining

84.6% of the errors the tagger made a wrong tag selection even though the word had been tagged with the correct tag in the training text.

This analysis indicates that further work should primarily be concentrated to improving the tagger's choice between already known word-tag combinations. Of course, an improvement of the tagging of one particular group of words also means a possible improvement of the tagging of other groups, since the choice made for one word is dependent on the choices for the surrounding words in the text.

Since only 3.9% of the tagging errors were due to previously unseen word-tags it is not motivated to allow other than known tags (for each particular word) to be selected when tagging known words (including tags not seen with the particular word in the training text). Allowing all tags to be selected would slow down tagging and would probably introduce almost as many, or more, errors than it would cure. However, if this assumption is actually true remains to be investigated.

By extending the SUC lexicon with SAOL words, as described in Implementation Considerations, the unknown words were reduced from 6.6% to 5.2%. The tagging accuracy improved from 96.3% to 96.4%. This is a rather modest improvement with respect to the cost; the lexicon size grows by a factor of four.

It is interesting to see if the language model we use allows the tagger to select highly lexically improbable tags for ambiguous words. In the sentence *Vad orsakar smärtan i min vad?* (What causes the pain in my calf?) the tagger correctly tags the first occurrence of *vad* as a pronoun and the second occurrence as a noun. This tagging is made despite the fact that *vad* was tagged as a pronoun 1850 times and as a noun only 3 times in the training text. Apparently, the contextual information has a strong enough impact on the model for the tagger to make a "daring" guess in this case.

Comparisons with other taggers

To be able to make relevant comparisons between different taggers, the same language, tag set, training and test texts should be used. Since there are no reported tagging results for the complete SUC that we know of, we can only estimate the capability of our tagger compared to others.

The Umeå tagger [15] uses a subset of 300 000 words of the SUC with a tag set of 194 tags and reports a tagging accuracy of 97.5%. The algorithm used is derived from the Volsunga algorithm [5] and uses the same type of statistical information as our algorithm, but unknown words are treated in another way. For unknown words, the Umeå tagger "cheats" and chooses only between tags that have been used to tag each unknown word in the *test text*. Since most new word types are tagged with only one tag in the test text, this way of treating unknown words corresponds to having almost 100% correct tagging of these words. Presumably, "hard" words are eliminated by the Umeå tagger in the same way as well. This discrepancy in approach means that tagging accuracy for the two taggers cannot be compared.

Tagging speed, however, can be compared. The Umeå tagger tags 500 words per second, compared with our tagger, which tags 14 000 words per second, both on a Sun Sparcstation 10.

Investigation of tagging errors

A manual inspection of the tagging errors of the optimization text words indicated that, of the 500 errors, 50% seemed very hard to tag correctly, given the language model used. Either a very long scope, understanding of the text, or other means of disambiguation are needed.

Disappointingly, 10% of the errors were caused by incorrect tagging in the supposedly correct SUC text. This fact indicates that the complete SUC contains 5 000 incorrect taggings, an error rate of 0.5%. Another 10% of the errors could be ascribed to inconsistent tagging in SUC or tagging errors in the training text. Too often, the same words and constructions were tagged in many different ways, without any obvious reasons. This observation gives at hand that tagging error rate could be reduced by 20% just by using a revised version of SUC. Correcting some errors in the test and training texts improved tagging accuracy to 97.0%.

The remaining 30% of the errors were considered within reach of tagging correctly. However, these errors were of many different types, and hence would require many different approaches for solving, each with a very small expected improvement. For example, the inclusion of wordtag-tag bigrams in Equation 4 gave a very modest improvement, if any, of the model.

Since the tagger will be used in a grammar checker, its performance on texts with grammatical errors is of interest. In order to test this performance, a test text with 200 sentences with agreement errors was compiled from authentic texts. Example: in *Den lilla huset* (The small house) the determiner *den* should be *det*. It was determined that 72% of the sentences were tagged in such a way that error rules could match the faulty construction.

The error rules at hand, from the old version of GRANSKA [23], detected 36% of the faults. By using the same rules without doing a syntactic disambiguation, as was done in old GRANSKA, only 22% of the faults were detected. Also, without syntactic disambiguation, 7 false alarms were generated, but none when the text was disambiguated.

Another test using 50 000 words from SUC and the same error rules gave 94 false alarms without disambiguation, but only 8 with disambiguation. Apparently, syntactic disambiguation is very important for grammar checking of Swedish text, as it increases the possibility to discover errors, as well as it reduces false alarms.

Discussion

The major inherent limitation of the Markov model is its short scope. A second order model cannot capture dependencies that span over more than three words. For example, to determine whether *verk* is singular or plural in *Violinkonserten är hans mest kända verk* (The violin concerto is his most known piece) would require a fifth order Markov model, since it depends on the number of the first word in the sentence only. But, as the required size of the training text grows exponentially in the order of the Markov model, there will never be a large enough training text to avoid the sparse data problem. Hence, other methods such as using correction rules are required.

The decision to employ a stochastic language model, however, has resulted in a versatile tagger, which, as mentioned earlier, can be used for most languages and tag sets and in most applications needing POS tagging. As we have not seen any reports on taggers with greater tagging speed than ours, we believe it is a state-of-the-art tagger in terms of speed. When we improve the tagger with error correction rules (see section Applications of the Tagger), we hope to construct a state-of-the-art tagger in terms of accuracy as well.

There exists a WWW version of our tagger, which anyone can test. The URL is `http://www.nada.kth.se/theory/projects/granska/`

Acknowledgments

The work has been funded by the Swedish research councils TFR, HSFR and Nutek.

Språkdata at Göteborg University and the Swedish Academy let us use Svenska Akademiens ordlista as a source for words in GRANSKA. Prof. Eva Ejerhed at Umeå University and Prof. Gunnel Källgren at Stockholm University let us use SUC. Ola Knutsson carried out the manual inspection of the tagging errors and evaluated the grammar checking performance.

References

[1] N. Ide and J. Véronis. Introduction to the special issue on word sense disambiguation: the state of the art. *Comp. Linguistics*, 24(1):1–40, 1998.

[2] E. Charniak, C. Hendrickson, N. Jacobson, and M. Perkowitz. Equations for part-of-speech tagging. In *11th National Conf. Artificial Intelligence*, pages 784–789, 1993.

[3] D. Cutting, J. Kupiec, J. Pedersen, and P. Sibun. A practical part-of-speech tagger. In *3rd Conf. Applied Natural Lang. Process.*, pages 133–140. ACL, 1992.

[4] E. Dermatas and G. Kokkinakis. Automatic stochastic tagging of natural language texts. *Comp. Linguistics*, 21:137–163, 1995.

[5] S. J. DeRose. Grammatical category disambiguation by statistical optimization. *Comp. Linguistics*, 14:31–39, 1988.

[6] J. Kupiec. Robust part-of-speech tagging using a hidden Markov model. *Comput. Speech Lang.*, 6:225–242, 1992.

[7] B. Merialdo. Tagging English text with a probabilistic model. *Comp. Linguistics*, 20:155–171, 1994.

[8] E. Brill. A simple rule-based part of speech tagger. In *3rd Ann. Conf. Appl. Natural Lang. Processing*, pages 152–155. ACL, 1992.

[9] A. Voutilainen. A syntax-based part-of-speech analyzer. In *7th Conf. European Chapter Assoc. Comp. Linguistics*, pages 157–164. ACL, 1995.

[10] J. Benello, A. W. Mackie, and J. A. Anderson. Syntactic category disambiguation with neural networks. *Comput. Speech Lang.*, 3:203–217, 1989.

[11] C. Samuelsson and A. Voutilainen. Comparing a linguistic and a stochastic tagger. In *35th Ann. Meeting Assoc. Comp. Linguistics*, pages 246–253. ACL, 1997.

[12] P. Tapanainen and A. Voutilainen. Tagging accurately – don't guess if you know. In *4th Conf. Applied Natural Lang. Process.*, pages 47–52. ACL, 1994.

[13] E. Charniak. *Statistical language learning*. MIT Press, Cambridge, Massachusetts, 1996.

[14] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Trans. Inf. Theory*, pages 260–269, 1967.

[15] M. Åström. A probabilistic tagger for Swedish using the SUC tagset. Technical report, Department of Linguistics, University of Umeå, Umeå, 1998.

[16] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical recipes*. Cambridge University Press, Cambridge, 1986.

[17] G. Grefenstette and P. Tapanainen. What is a word, what is a sentence? Problems of tokenization. In *3rd Int. Conf. Comp. Lexicography*, pages 79–87, 1994.

[18] 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.

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

[20] V. Kann, R. Domeij, J. Hollman, and M. Tillenius. Implementation aspects and applications of a spelling correction algorithm. In R. Koehler, L. Uhlirova, and G. Wimmer, editors, *Text as a Linguistic Paradigm: Levels, Constituents, Constructs. Festschrift in honour of Ludek Hrebicek*. Universitat Verlag, Trier, Germany, 1999. To appear. Available on WWW from `http://www.nada.kth.se/theory/projects/swedish.html`.

[21] 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. The web page of SUC is `http://www.ling.su.se/DaLi/Projects/SUC/`.

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

[23] R. Domeij, O. Knutsson, S. Larsson, K. Severinsson-Eklundh, and Å. Rex. Granskaprojektet 1996–1997. Technical Report IPLab-146, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, 1998.