Växjö University

School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

A Probabilistic Part-of-Speech Tagger with Suffix Probabilities

Johan Hall

April
2003

MSI
Växjö University
SE-351 95 VÄXJÖ

Report xxxxx
ISSN xxxx-xxxx
ISRN VXU/MSI/DA/E/--xxxxx/--SE

A Probabilistic Part-of-Speech Tagger with Suffix Probabilities

Johan Hall
Master's Thesis

April 2, 2003

Abstract

This master's thesis deals with the problem of automatic part-of-speech tagging, where the main task is to assign the correct part-of-speech to each word in a text.A part-of-speech tagger is used in many language technology applications as a first step in a longer process such as grammar checking, machine translation, information extraction and information retrieval.

The part-of-speech tagger developed in this thesis is based on probability theory. The two main probabilistic models used are the lexical model and the contextual model. To estimate the probabilities the tagger uses supervised machine learning and trains the model using a collection of texts called corpus.

This thesis defines and empirically evaluates models based on suffix probabilities for dealing with the problem of unknown words part-of-speech tagger. Furthermore, it describes an efficient implementation of a part-of-speech tagger with suffix probabilities, using a trie data structure. Around 95% accuracy rate is reported and the part-of-speech tagger tags about 70000 tokens in a second.

Contents

1 Introduction **2**

2 Background **4**
2.1 Part-of-Speech Tagging 4
2.2 Probabilistic Part-of-Speech Tagging 6
2.3 Hidden Markov Models 7
2.3.1 Lexical model 7
2.3.2 Contextual model 8
2.4 Smoothing 8
2.4.1 Additive Smoothing 8
2.4.2 Linear Interpolation 9
2.4.3 Choice of smoothing method 9
2.5 Viterbi 9

3 Tagging models **11**
3.1 Baseline model 11
3.2 Lexical models 13
3.3 Contextual models 16

4 Implementation **18**
4.1 Preliminaries 18
4.2 Lexical model 20
4.3 Contextual model 21
4.4 Segmentation and Viterbi 22
4.5 Tokenizer 24
4.6 Tagset 24
4.7 Options 24

5 Evaluation **26**
5.1 Corpus data 26
5.2 Evaluation 26
5.3 Results 27
5.3.1 Lexical model 27
5.3.2 Contextual model 29
5.3.3 Final test 31
5.4 Discussion 32

6 Conclusion **34**

A The SUC small tagset **37**

B The SUC large tagset **38**

Chapter 1

Introduction

This master's thesis deals with a common problem in the area of language technology: How can we automatically assign parts-of-speech to words in a text? Examples of parts-of-speech are noun, adjective and verb. The result of a part-of-speech tagging maybe in itself is not so interesting, but there are many applications in language technology where this information is useful. For example, a grammar checking application maybe needs to know the part-of-speech of each word as a first step in the analysis process.

At first sight this problem may sound trivial, but it is actually very hard to solve. In fact there is no known method that solves the problem with complete accuracy and it may well be impossible to assign a part-of-speech to each word in a text correctly. There are studies which have tested two humans with linguistic training on this task, and which have found that they disagree on 2 percent of the words [13]. It is therefore reasonable to restrict the problem a little. How can we automatically assign parts-of-speech to words in a text with the highest possible accuracy rate?

In this thesis I define, implement and empirically evaluate a probabilistic solution to the part-of-speech tagging problem for Swedish text. This means that I need to define and implement probabilistic models to cope with the problem. A big help in this approach is to have access to tagged text, which means that every word and punctuation mark in the text has information about its part-of-speech, or lexical category. A large collection of such text is called corpus and I will use the Stockholm-Umeå Corpus (SUC) of written Swedish in this work [6, 10]. I will in this report use the term *tag* which refers to a lexical category or part-of-speech in the context of a particular corpus. The term *tagset* will also be used and refers to the set of all tags which the part-of-speech tagger can use to tag a text. There is no standard tagset for Swedish so I will use the same tagsets that the Stockholm-Umeå Corpus (SUC) uses.

The purpose of this master's thesis is to develop part-of-speech tagger for Swedish which is both accurate and efficient. I will investigate if a probabilistic model with word suffixes in combination with well known models can be used to achieve the complementary goals of accuracy and efficiency.

My primary goal is to define probabilistic models which result in a 95 percent accuracy when tagging untagged text with the Stockholm-Umeå Corpus. These models should be implemented in such a way that the tagger is able to tag 20000 tokens per second on average.

Hopefully the tagger will be used in the future and this means that the tagger must be easy to change and extend. A secondary goal when implementing the tagger is to design and implement it in such a way that new modules can be inserted easily, such as a new tokenizer or the support of new corpus resources. It is also important that the tagger is easy to use.

The primary goals will be carefully evaluated in chapter 5. The secondary goals will not be evaluated in the strict sense of doing user tests. How the secondary goals are accomplished will be discussed in chapter 4. The implemented part-of-speech tagger is called MALT tagger after the name of the group MALT (Models and Algorithms for Language Technology) at Växjö University working with language technology.

The thesis is organized as follows. Chapter 2 presents the background theory which explains the models and algorithms needed. Chapter 3 explains the models used in the practical work of the thesis. Implementation of the part-of-speech tagger is presented in chapter 4. Empirical evaluation of the models explained in chapter 3 is presented in chapter 5. Finally, chapter 6 is devoted to the main conclusions of the study.

Chapter 2

Background

2.1 Part-of-Speech Tagging

The task of assigning to every word in a text a part-of-speech like noun or verb is called part-of-speech tagging. Consider the following Swedish sentence:

```
Mot slutet av sommaren 1988 hade hon sina första egna
långritter genom smålandsskogarna.
```

The output of a part-of-speech tagger is a sequence of parts-of-speech attached to every word and punctuation mark. In table 2.1, we can see that every *token* i.e.

Table 2.1: Example of tagged text

Word	**Part-of-speech**	**Morphosyntactic properties**
Mot	PP	
slutet	NN	NEU SIN DEF NOM
av	PP	
sommaren	NN	UTR SIN DEF NOM
1988	RG	NOM
hade	VB	PRT AKT
hon	PN	UTR SIN DEF SUB
sina	PS	UTR/NEU PLU DEF
första	RO	NOM
egna	JJ	POS UTR/NEU PLU IND/DEF NOM
långritter	NN	UTR PLU IND NOM
genom	PP	
smålandsskogarna	NN	UTR PLU DEF NOM
.	MAD	

word or punctuation mark, has a tag attached to it, listed in the second column of the table. Each tag refers to a part-of-speech and the tagset used here, as well as the example sentence, is taken from the Stockholm-Umeå Corpus [6, 10]. A *corpus* is a collection of texts from different areas such as newspaper text

and scientific articles. A corpus in most cases contains extra information about every word such as its part-of-speech and morphosyntactic properties. Often the tagset is extended to include also morphosyntactic properties such as number and gender for nouns, which is what we find in the third column in Table 2.1. The full tagsets with and without morphosyntactic features are described in appendix A and B.

Part-of-speech tagging can be used in many applications as one step in a longer process, example in machine translation, information retrieval, information extraction and grammar checking. If we want to translate the Swedish word *klippa* to English, the word can be translated as either *cut* or *cliff* in English. If we know that the word *klippa* is a verb the translation is easier: *cut*. Information extraction applications are using patterns for extracting information from text and often make reference to parts-of-speech in templates [2].

Because a natural language like Swedish contains many ambiguous words regarding parts-of-speech, it is hard or impossible to construct a tagger that tags with an accuracy of 100 percent. Consider the following Swedish sentences:

```
Jag ska resa till Australien.
Min resa till Australien var underbar.
```

The word *resa* has two possible parts-of-speech depending on context: noun or verb. In the first sentence the word is a verb and in the second sentence the word is a noun. Even if we list all words in a language with their possible parts-of-speech we still have the problem with ambiguous words. When we construct a part-of-speech tagger we must obtain information about the context and then develop a method of disambiguating these hard words.

A natural language has evolved as a means for humans to communicate with each other and a language is usually open in the sense that it develops over time. A natural language is in constant evolution. Thus, in addition to the basic ambiguity the following factors make part-of-speech tagging difficult:

- *Unknown words*: Over time we can not expect to have a complete list of every word in a natural language. New words become popular, other words will be forgotten and other words will get a new spelling. We must find a solution for handling unknown words.
- *Indeterminacy*: Since a natural language is an open and evolving system, there is sometimes disagreement about which tag is the correct one even among human experts.
- *Noise*: There are also errors in the data resources which help us build language technology applications.

There are parts-of-speech which usually only contain a small number of words, such as conjunctions and prepositions, and these parts-of-speech are known as closed classes. It is not likely that new words will be added frequently to the close classes. There are however closed classes which contain a large number of words such as numerals. The opposite of closed classes are the open classes containing parts-of-speech which usually have thousands of members and where new members are added continuously, such as nouns and verbs. The distinction closed and open classes is relevant when handling unknown words, since these are more likely to belong to open classes [9].

There are several different approaches to solving the problem of part-of-speech tagging:

- *Rule-based part-of-speech tagging*: The first attempts to do part-of-speech tagging were based on a two step approach. In the first step every word was assigned a list of potential parts-of-speech from a dictionary. In the next step the text was processed by a list of hand-written disambiguation rules. After applying every rule the list was reduced to only containing one part-of-speech for each word [9, 2].

- *Probabilistic part-of-speech tagging*: This approach uses probabilistic models to determine the most probable part-of-speech sequence for a given word sequence. The models usually incorporate probabilities of words in relation to the parts-of-speech and of parts-of-speech in relation to other parts-of-speech [9].

- *Transformation-based part-of-speech tagging*: Transformation-based, tagging often also called Brill tagging, combines the benefits of both rule-based and probabilistic parts-of-speech tagging. Usually the tagger first assigns to every word the most likely part-of-speech. This approach will introduce several errors. The next step is to correct as many errors as possible by applying transformation rules that the tagger has learned [2].

2.2 Probabilistic Part-of-Speech Tagging

A popular solution is known as probabilistic part-of-speech tagging and uses a probabilistic model during the tagging phase. The tagger tries to find the most probable part-of-speech sequence for a given word sequence. In most cases the probabilistic model consists of a lexical and a contextual model. In the lexical model, every word has a set of probable parts-of-speech. However, only using a lexical model is not enough to tag with high accuracy. We also need a model to capture the context of tags. In the contextual model, every part-of-speech is conditioned on its neighboring parts-of-speech.

We can use supervised machine learning to generate the lexical and the contextual model. To do so we need manually annotated data. In other words, we need corpus where a part-of-speech is attached to every token. One way to use the corpus is to count every occurrence of a particular word tagged with a particular tag and use this frequency to compute probabilities for the lexical model. By counting the occurrences of a particular in the context of other tags, we can compute corresponding frequencies for the contextual model. If no tagged corpus is available from which we can obtain the probabilities needed, we can use unsupervised learning. There is an algorithm called Baum-Welch that is suitable when no tagged corpus is available [2]. Whether we use supervised or unsupervised learning the corpus used to derive probabilities is referred to as the training corpus.

Supervised learning in most cases is the best way to obtain the probabilities needed. Merialdo [14] suggests that the optimal strategy to build the best possible model for tagging is the following:

- Use as much tagged text as possible.

- Compute the relative frequencies to obtain an initial model.
- Collect as much untagged text as possible.
- Use the Baum-Welch algorithm on the untagged text in several iterations. If the tagging quality degrades when using the new model with untagged text stop using it.

2.3 Hidden Markov Models

Markov models are state-space models that can be used to model a sequence of random variables that are not necessarily independent. The most common model used by probabilistic taggers is the n-class model which can be implemented as a hidden markov model (HMM) with states corresponding to sequence of parts-of-speech and output symbols corresponding to tokens [14]. In a HMM, we do not know the state sequence that the model passes through, but only the output sequence which is a probabilistic function of it. In a part-of-speech tagger application we define the problem more formally as follows. For a given string of words $w_1, ..., w_k$, HMM taggers generally find the tag sequence $t_1, ..., t_k$ which maximizes the following probabilistic function [9]:

$$\prod_{i=1}^{k} P(w_i|t_i)P(t_i|t_{i-(n-1)}, ..., t_{i-1}) \tag{2.1}$$

The first factor of this product is called the *lexical model* and the second factor is called the *contextual model* (see section 2.3.1 and 2.3.2). When we have computed the necessary probabilities we can implement the tagger as a HMM (see section 2.5).

2.3.1 Lexical model

The primary purpose of the lexical model is to have a lexicon with possible tags for each word in the training data. If the lexical model is to be useful it must also give the probability $P(w|t)$, in formula (2.1), for every word w and tag t. This probability can be estimated with relative frequencies:

$$P(w|t) = \frac{C(w,t)}{C(t)}$$

If $C(u)$ is the number of times an outcome u occurs in N trials then $C(u)/N$ is the relative frequency of u [12]. To obtain the relative frequencies for the lexical model we simply count every word with a specific tag and divide it with the number of occurrences for this particular tag, which gives the conditional probability of the word given the tag. For example the Swedish word *resa* will have two probabilities $P(resa|NN)$ and $P(resa|VB)$ (where NN and VB denotes the classes noun and verb respectively).

However, if a word is not included in the training data the probability will be estimated with the value zero and this will normally decrease the tagger's performance. Handling the problem of unknown words is one important key to increasing the performance of the part-of-speech tagger.

One way of handling the problem with unknown words is to find the suffixes of words and include them in the model. Carlberger and Kann [3] has explored word-endings as one solution to the problem and found that the tagging accuracy of unknown words increased with increasing length of the suffix, but no significant improvement was detected for a suffix length longer than 5. The use of suffix probabilities for tagging of unknown words has also been explored by Samuelsson [16], who found that a suffix length of approximately 4 letters gave the best performance for unknown word estimation depending on the corpus size.

2.3.2 Contextual model

Only relying on a lexical model is not a good idea because the part-of-speech tagger will not have information about the context where the word is found. The probability $P(t_i|t_{i-(n-1)}, ..., t_{i-1})$, in formula (2.1), for every $t_{i-(n-1)}, ..., t_{i-1}$ can be estimated with relative frequencies:

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

Varying the value of n will generate different n-class models. In part-of-speech tagging usually n is equal to 2 or 3, and the resulting models are referred to as bigram and trigram models, respectively. The choice of value for n is dependent on the training data. Nivre [15] has found that the biclass model is preferable for large tagsets and/or small training corpora because of sparse data.

2.4 Smoothing

A common problem with probabilistic part-of-speech taggers is that the amount of training data is not sufficient, and estimation is not reliable due to sparse data. To tackle this problem the model needs a mechanism to improve probability estimates for rare events. This mechanism is often called *smoothing*. The term smoothing describes techniques for adapting the relative frequency estimates to get more accurate probabilities [4]. If we do not eliminate zero frequencies, they are likely to generate errors. There are many methods for smoothing such as additive smoothing, Good-Turing estimation and back-off smoothing [15]. Below I will concentrate on the methods used in this thesis.

2.4.1 Additive Smoothing

One of the simplest methods of smoothing is called *additive smoothing*. The method consists in adding a constant k to every frequency to avoid zero frequencies:

$$P_{add}(x) = \frac{C(x) + k}{N + kN_X}.$$

where x is the value of a stochastic variable X, where x has the observed frequency $C(x)$ in a sample of N observations, and where X has a sample space of N possible values. Often the constant k is assigned the value 0.5 according to Lidstone's Law [4].

2.4.2 Linear Interpolation

Another way of smoothing is *linear interpolation*. Here we try to combine several n-gram models by using a weighted sum of available models. If we want to smooth the trigram probability we interpolate additive smoothed unigram, bigram and trigram by using following formula:

$$P_{int}(t_i|t_{i-2}, t_{i-1}) = \lambda_1 P(t_i) + \lambda_2 P(t_i|t_{i-1} + \lambda_3 P(t_i|t_{i-2}, t_{i-1}),$$

the optimization parameters λ_1, λ_2 and λ_3 must have a sum of 1 [3]. To find the best values of the optimization parameters is the main problem with linear interpolation. We can experiment with different values, but it is hard to find the optimal values.

2.4.3 Choice of smoothing method

The use of more sophisticated smoothing methods such as Good-Turing estimation often outperforms simple additive smoothing, as has been reported by Gale and Sampson [7] and Nivre [15]. In this thesis I will investigate whether the use of suffix probabilities for handling unknown words can eliminate the need for more complex smoothing method for the lexical model.

With respect to the contextual model, Nivre [15] has shown that both additive smoothing and back-off smoothing outperforms Good-Turing estimation. In this thesis, I will also explore linear interpolation, which was not among the methods evaluated by Nivre [15] but which has been found to work well for the contextual model by Carlberger and Kann [3].

2.5 Viterbi

If we have a particular word sequence and a Hidden markov model with states represents tags, which path is the most probable through the model? A time consuming way to answer this question is to calculate every possible path and pick the most probable path. The problem with this approach is that the time complexity is exponential in the length of the input sequence. Another solution is to use dynamic programming which solves problems by combining the solutions of subproblems. A dynamic programming algorithm solves the problem by saving results from subproblems in a table, thus avoiding re-computation every time the subproblem occurs [5].

The Viterbi algorithm uses dynamic programming for finding the optimal solution [17]. The Viterbi algorithm reduces the complexity of the problem of finding the best state sequence to polynomial time and the algorithm is linear in the number of words to be tagged.

Viterbi exploits the fact that the probability for being in state q at time i only depends on the state of the model at time $i - 1$. The algorithm saves the highest probability going from a state to the next state in a matrix, called a trellis. In one dimension we have the word sequence with $|S| + 1$ items (where S is the sequence tokens in the input sequence) and in the another dimension of the matrix we have the possible states with $|T| + 1$ states if we use the bigram model (where T is the set of distinct tags). The extra state is a special initial state. The trigram model will have $(|T|+1)^2$ states. For each word the algorithm

fills a column in the trellis matrix. It calculates the probability for all possible transitions from previous states to current state. If the new score is better it is saved in the matrix. At the same time the algorithm saves the state in another matrix called back-pointer. The task of this matrix is to keep track of the best path. When the whole matrix is filled we need to find the most probable path by finding the highest value in the last column. Then we use this state to find the path in the back-pointer matrix [9].

The algorithm has a linear time complexity in the number of words to be tagged, but quadratic in the number of states. Since running time is also affected by the efficiency of with which lookup of lexical and contextual model can be performed, there is considerable room for optimization even when the Viterbi algorithm is used.

Chapter 3

Tagging models

In order to develop an accurate tagger for Swedish I have experimented with different variants of the lexical and contextual models described in chapter 2. In this chapter, I will define the different models used. In the next chapter I will describe their implementation and in chapter 5 I will present an empirical evaluation of different models for tagging Swedish text.

3.1 Baseline model

First I will define a baseline model for both the contextual and lexical model which will be the starting point of the comparison between different models. The baseline contextual model (CM1) uses additive smoothing to smooth the different n-class models based on the relative frequencies. Maybe the smoothing of the unigram is not needed because it is not likely that there is any problem with sparse data, but I do it anyway to get a homogeneous model.

$$P_{CM1}(t_i) = \frac{C(t_i) + k}{N + k(|T| + 1)}$$

$$P_{CM1}(t_{i-1}, t_i) = \frac{C(t_{i-1}, t_i) + k}{(N-1) + k(|T| + 1)^2}$$

$$P_{CM1}(t_{i-2}, t_{i-1}, t_i) = \frac{C(t_{i-2}, t_{i-1}, t_i) + k}{(N-2) + k(|T| + 1)^3}$$

The unigram uses the frequency $C(t_i)$ which is the number of occurrences of the tag t_i in the training data $C(t_{i-1}, t_i)$ is the number of occurrences of the tag bigram (t_{i-1},t_i), that is, the tag t_i preceded by the tag t_{i-1}. Finally we have the frequency $C(t_{i-2}, t_{i-1}, t_i)$ which is the number occurrences of the tag trigram (t_{i-2}, t_{i-1}, t_i), that is, the tag t_i preceded by the tags t_{i-2} and t_{i-1} in that order.

The number of tokens in the training data is denoted by N. Since the bigram model covers two tokens and the trigram model of three tokens in the training data we will only have $N - 1$ and $N - 2$ observations of bigrams and trigrams, respectively. The constant k will be assigned the value 0.5. The set T is defined as the set of tags and in the denominator there is a term $|T|$ which is the number of tags in the tagset. We add one because we have a special case

when the part-of-speech tagger starts to tag a new string of words. In section 4.4 we assume that this artificial start tag, occurring before the first word in the string has the same distribution as the delimiter tag MAD. We will have $(|T|+1)^2$ combinations in the bigram model and $(|T|+1)^3$ combinations in the trigram model. The following

$$P_{CM1}(t_i|t_{i-1}) = \frac{P_{CM1}(t_{i-1}, t_i)}{P_{CM1}(t_{i-1})}$$

$$P_{CM1}(t_i|t_{i-2}, t_{i-1}) = \frac{P_{CM1}(t_{i-2}, t_{i-1}, t_i)}{P_{CM1}(t_{i-2}|t_{i-1})}$$

equations give the conditional probabilities for bigrams and trigrams based on the additive smoothing. These probabilities are the ones used in the contextual model.

The lexical model is used to estimate the probability of a word given its part-of-speech $P(w|t)$. The key to developing an accurate probabilistic part-of-speech tagger is to handle the problem of sparse data. What should the part-of-speech tagger do when there is not enough data to do a reliable estimates. The baseline lexical model (LM1) will handle this problem with additive smoothing.

$$P_{LM1}(w, t) = \frac{C(w, t) + k}{N + k|W|}$$

The frequency $C(w, t)$ is the number of occurrences of the word w tagged with t in the training data. The probability $P_{LM1}(w, t)$ for all known words will be estimated by adding the constant $k = 0.5$ in the numerator to the number of occurrences of the word w tagged with t in the training data. In the denominator the same constant $k = 0.5$ is multiplied with the number of word types $|W|$ in the training data and then added to the number of tokens N in the training data. When a token can not be found in the lexicon we will treat this token as an instance of the single unknown word type w_u [15]. The conditional probabilities used in the lexical model are defined as follows:

$$P_{LM1}(w_u, t) = \frac{k}{N + k|W|}$$

$$P_{LM1}(w|t) = \frac{P_{LM1}(w, t)}{P_{CM1}(t)}$$

$$P_{LM1}(w_u|t) = \frac{P_{LM1}(w_u|t)}{P_{CM1}(t)}$$

The tagger must review every tag when a token can not be found in the lexicon. To improve the baseline model we can eliminate all closed classes of parts-of-speech. If the denote open classes as OC we can define OC as the set of tags:

$$OC = \{t \in T \mid \frac{|W_t|}{\sqrt{C(t)}} > \epsilon\}$$

This equation is based on the type/token ratio $|W|/\sqrt{N}$ when the corpus grows [8]. We can use type/token ratio to determine if the tag t is a open class. When the value of the type/token ratio is below ϵ we can consider the tag t as a closed class, otherwise its an open class. I will use the constant $\epsilon = 10$.

All known word types in the training data is a set W and we summarize the probability P_{LM1} for the baseline lexical model:

$$P_{LM1}(w|t) = \begin{cases} P_{LM1}(w|t) & \text{if } w \in W \\ P_{LM1}(w_u|t) & \text{if } w \notin W \wedge t \in OC \\ 0 & \text{otherwise} \end{cases}$$

3.2 Lexical models

In this thesis I will study the effect using a lexical model with suffix probabilities, I will call this the suffix model. First we need to define what a suffix is in this context. A word can be built up from smaller units called morphemes. There are two basic classes of morphemes: stems and affixes.[1] The stem is the part that provides the main meaning of a word while affixes contribute to this meaning. Affixes are divided into smaller categories. One of the category is the class suffixes which follow the stem of a word [9].

In this thesis I do not use the term suffix in the proper sense of a morpheme following the stem because it is hard in the training process to determine which suffixes a word has. Instead I use suffix in the sense of string suffix and say that X is a suffix of Y iff there is a string Z such that $ZX = Y$.

Thus, suppose that the Swedish word *sill* has a positive probability to be a noun, i.e. $P(sill|\text{NN}) > 0$. Then the following probabilities should also be positive $P(-ill)|\text{NN})$, $P(-ll)|\text{NN})$ and $P(-l)|\text{NN})$.

These probabilities should have values that reflect the relative share of each suffix in the class of nouns. If we use an appropriate data model for representing words and their probabilities, we should be able to look up suffix probabilities for unknown words efficiently by the same operation that is used for the lexical probability of known words. For this reason, I will use a trie, or letter tree, to represent the lexical model [1]. Moreover, I will insert words backwards in order to facilitate the retrieval of suffixes.

Figure 3.1 shows an example of a letter tree with the Swedish words *in*, *bin*, *din*, *fin*, *kusin*, *latin* and *att* inserted in the letter tree backwards. If a node has an asterisk it means that the path from the node to the child of root is a word, given the small amount of training data enumerated above. When we want to use the letter tree to find a token we must traverse the trees starting from the end of the word. We can define four cases when searching for a token in a lexicon organized as a backwards letter tree:

1. The traversal of the letter tree is successful according to the token and the final node in the traversal is a word. This means that the token is a known word. This happens, for example, when we look up the words *bin*, *att* and *kusin* in the tree in figure 3.1.

2. The traversal of the letter tree can not be completed, but the node where the traversal stops is a word. This happens in our example when look up the words such as *rabbin* and *kabin*, since *bin* is a word in the training data.

[1] Strictly speaking stems may be decomposed into roots, but this complication is ignored.

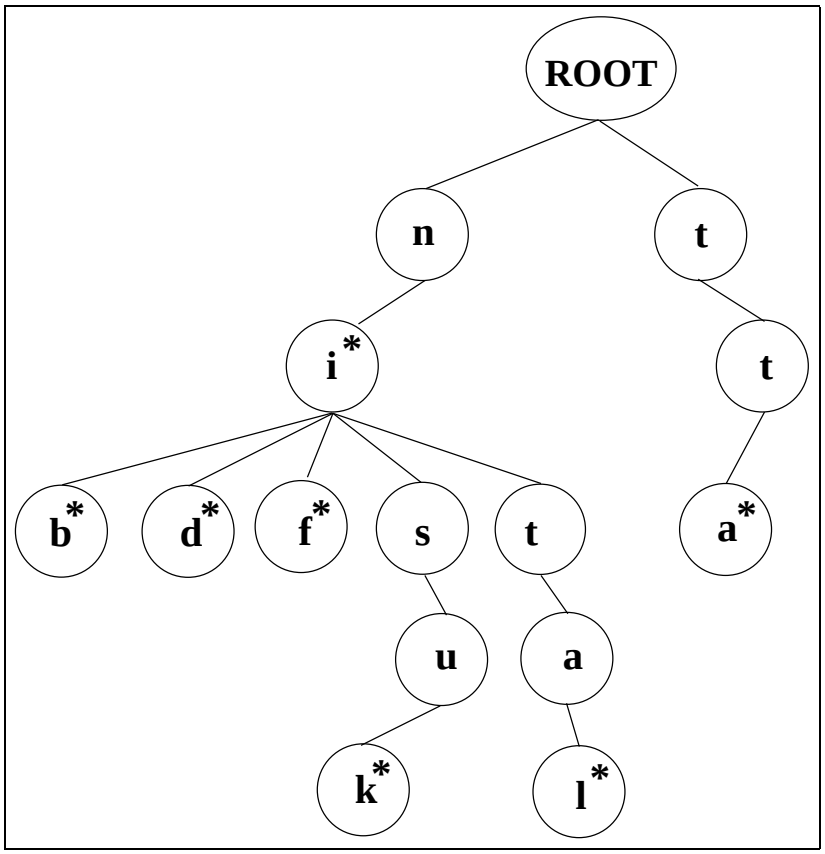

Figure 3.1: Letter tree

3. The traversal of the letter tree is successful, but the node is not a word. This happens when we look up the word *sin*. Note that the word *sin* is a Swedish word, but it is not a word in the training data.

4. The traversal of the letter tree can not be completed and the node is not a word. This occurs when we look up a word such as *kantin*, since the longest known suffix *-tin* is not a word in the training data.

Imagine now that we have access to a large corpus containing about one million word tokens with the proper tags and we use this material to build up the letter tree. In each node there are two lists, one containing the frequency of each tag with the given string as a suffix of longer word, and one containing the frequency of each tag with the given string as a complete word. Furthermore, we will keep track of whether a token is capitalized or not. In this way we can easily build up a lexical model with word and suffix probabilities.

Below I will define different lexical models with suffix probabilities. A lexical model (LM) will generally be defined in terms of two more specific models, a lexical model for known words (LW) and a lexical suffix model (LS).

The first lexical model (LM2) has two cases: when a token can be founded in the lexicon and when it can not be found.

$$P_{LW2}(w|t) = \frac{P_{LM1}(w,t)}{P_{CM1}(t)}$$

If the token is known we simple use the same probability as we did in the baseline model. This means that a token must be traversed successfully in the letter tree and the final node in the traversal must be a word; otherwise the token is unknown. If the token is unknown we use the longest known suffix s to calculate the suffix probability:

$$P_{LS2}(s,t) = \frac{C(s,t)+k}{N+k|W|}$$

$$P_{LS2}(s|t) = \frac{P_{LS2}(s,t)}{P_{CM1}(t)}$$

$$P_{LM2}(w|t) = \begin{cases} P_{LW2}(w|t) & \text{if } w \in W \\ P_{LS2}(s|t) & \text{otherwise} \end{cases}$$

In the general suffix model I use the longest possible suffix s which can be found in the letter tree.

The lexical models (LM3-LM6) are variants of the lexical model (LM2), where I will reuse the definitions of $P_{LW2}(w|t)$ and $P_{LS2}(s|t)$ if possible. The only difference between lexical models (LM2) and (LM3) is that a token could also use probability $P_{LW2}(s|t)$ if the longest suffix s is a known word. This means that the final node in the traversal of the token is a word, otherwise the token is unknown.

$$P_{LM3}(w|t) = \begin{cases} P_{LW2}(w|t) & \text{if } w \in W \\ P_{LW2}(s|t) & \text{if } w \notin W \wedge s \in W \\ P_{LS2}(s|t) & \text{otherwise} \end{cases}$$

In the lexical baseline model we eliminated closed classes as candidate parts-of-speech for unknown words. With the lexical model (LM4) we will see if the performance increases when we eliminate closed classes using the same set of open classes OC:

$$P_{LM4}(w|t) = \begin{cases} P_{LW2}(w|t) & \text{if } w \in W \\ P_{LS2}(s|t) & \text{if } w \notin W \wedge t \in OC \\ 0 & \text{otherwise} \end{cases}$$

Usually a Swedish sentence starts with a capitalized word. With the lexical model (LM5) I will define a model which takes advantage of this knowledge. If a token comes after a delimiter in the set $D = .,!,?$ occurrences we summarize the frequencies of capitalized and non-capitalized occurrences when calculating the probabilities. We can define the model P_{LM5} as follows:

$$P_{LW5}(w,t) = \frac{C(w_{lower},t) + C(w_{upper},t) + k}{N+k|W|}$$

$$P_{LW5}(w|t) = \frac{P_{LW5}(w,t)}{P_{CM1}(t)}$$

$$P_{LS5}(s,t) = \frac{C(s_{lower},t) + C(s_{upper},t) + k}{N+k|W|}$$

$$P_{LS5}(s|t) = \frac{P_{LS5}(s,t)}{P_{CM1}(t)}$$

where w_{lower} is the same as w except (possibly) that the first letter is in lowercase, and w_{upper} is the same as w except (possibly) that the first letter is in uppercase. The notation s_{lower} and s_{upper} is used analogously except that "the first letter" is the first letter not of the suffix but of the complete word of which it is a part.

$$P_{LM5}(w|t) = \begin{cases} P_{LW2}(w|t) & \text{if } w \in W \wedge w_{i-1} \notin D \\ P_{LS2}(s|t) & \text{if } w \notin W \wedge w_{i-1} \notin D \\ P_{LW5}(w|t) & \text{if } w \in W \wedge w_{i-1} \in D \\ P_{LS5}(s|t) & \text{if } w \notin W \wedge w_{i-1} \in D \end{cases}$$

Finally, we will see if a different maximum suffix length can improve the performance of the tagger. If the token is known in the lexicon the probability $P_{LW2}(w|t)$ will be used, but when the token is unknown a different suffix model will be used.

$$P_{LM6}(w|t) = \begin{cases} P_{LW2}(w|t) & \text{if } w \in W \\ P_{LS2}(s_L|t) & \text{otherwise} \end{cases}$$

The suffix s_L is longest possible suffix but not longer than L. This means that if a token can not be traversed completely in the letter tree or the final node of the token is not a known word and the path from the root's child is longer than L, then we use the suffix with path length L.

3.3 Contextual models

In section 2.3.2, we saw how the contextual probabilities can be computed with relative frequencies in different n-class models. Without any smoothing we can assume that the tagger will not perform well because there are probably zero frequencies in the training data. For the baseline contextual model (CM1) in section 3.1 we defined different n-class models with additive smoothing.

The contextual model (CM2) and (CM3) instead use linear interpolation, see section 2.4.2. First, linear interpolation is computed with relative frequencies (CM2).

$$P_{CM2}(t_i|t_{i-1}) = \lambda_1 \frac{C(t_{i-1})}{N} + \lambda_2 \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}$$

$$P_{CM2}(t_i|t_{i-2}, t_{i-1}) = \lambda_1 \frac{C(t_{i-1})}{N} + \lambda_2 \frac{C(t_{i-1}, t_i)}{C(t_{i-1})} + \lambda_3 \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})}$$

Secondly, linear interpolation is computed with additive smoothing (CM3). I reuse the definition of unigram $P_{CM1}(t_i)$, bigram $P_{CM1}(t_i|t_{i-1})$ and trigram $P_{CM1}(t_i|t_{i-2}, t_{i-1})$ in the baseline model (CM1).

$$P_{CM2}(t_i|t_{i-1}) = \lambda_1 P_{CM1}(t_i) + \lambda_2 P_{CM1}(t_i|t_{i-1})$$

$$P_{CM3}(t_i|t_{i-2}, t_{i-1}) = \lambda_1 P_{CM1}(t_i) + \lambda_2 P_{CM1}(t_i|t_{i-1}) + \lambda_3 P_{CM1}(t_i|t_{i-2}, t_{i-1})$$

The optimization parameters λ_1, λ_2 and λ_3 are different in the bigram and trigram model, but I will use the same optimization parameters for the model

computed with relative frequencies (CM2) and with additive smoothing (CM3). I will determine the optimization parameters by simply experiment with different values and use the value which gives the highest accuracy.

Chapter 4

Implementation

Now we have the contextual and lexical models defined in chapter 3. Before an empirical evaluation of the models can be done we need a program which implements the models in an efficient way. In this chapter I will present the general design and the most important implementation issues. In chapter 1 I formulated primary and secondary goals about what the tagger should accomplish. In section 4.1 discuss shortly how these goals are accomplished.

4.1 Preliminaries

The obvious goal for a part-of-speech tagger is to tag as many words correctly as possible. By using well-known theory about probabilistic tagging and the defined models in the previous chapter, we can hopefully get a fairly high accuracy rate. Chapter 5 is dedicated to a discussion and evaluation of the models, so in this chapter I will only explain how the models implemented.

Another important goal are efficiency regarding time and space consumption. The first step to better performance is to improve the algorithm. I have tried to eliminate unnecessary loops, move out code from loops and reduce the number of iterations. I have gained the must of the improvements with a sharpened algorithm. This work is described in section 4.4. It is also important to optimize the code using a good compiler with all available optimization features.

If an application is not user friendly it may not be used. The target user of the application is user with knowledge about programming who knows how to compile a program. I have created a console-based user interface. The application can be executed with different options. To handle these options the user only has to change a text file before executing the application. This is described in section 4.7. In the future I will maybe do a graphical user interface as a web-based application.

Hopefully the tagger will be used in the future and one of the secondary design goals was to design and implement the tagger in such a way that it can be extended and changed easily. I have created an application with exchangeable modules so that new module can be added in the future e.g. a new tokenizer. Figure 4.1 gives a schematic view of the MALT tagger. The ellipses are programs, the boxes are modules in the programs and to the right there is a logical view of secondary storage. Some of the smaller boxes corresponds to implemen-

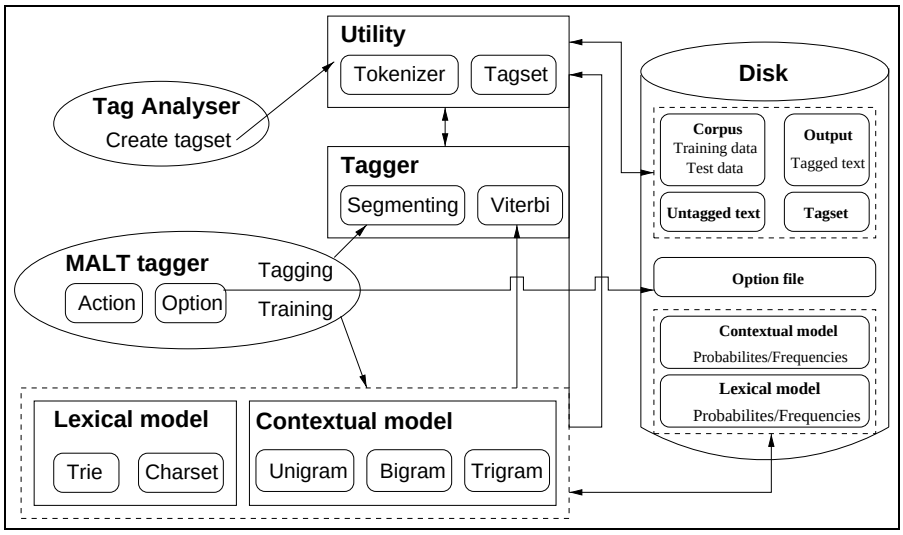

Figure 4.1: Schematic view of the part-of-speech tagger

tation units, but not all of them. The arrows between the shapes indicate a interaction.

As already mentioned MALT tagger can be executed in different ways by editing an option file. The option and action modules handle how the tagger should be executed. The two main uses of the tagger are training and tagging.

The training phase builds the contextual and lexical models given a corpus of training data and is implemented in the module with the same name. The lexical model uses a trie data structure to store suffixes and words. The contextual model uses n-dimensional arrays depending on which n-class model the array is intended for. The contextual and lexical models are stored in separate files in the secondary storage after training.

The tagging phase loads contextual and lexical models from file and then the tagger module takes over. The tagger module have two main tasks, to divide the untagged text into chunks and then tag each chunk using the Viterbi algorithm.

The utility module contains auxiliary functions for the two main phases. The tagset unit helps the program convert between the internal representation of the tagset and the currently used tagset. The tokenizer unit helps the program to read tagged and untagged text such as different corpora or ordinary text in special formats. The tag analyzer is an auxiliary program which generate tagset files with the internal representation of the tagset from different corpora.

Which programming language should I use? It is not easy to choose programming language from a scientific point of view. Here are some of the issues involved:

- Which languages do you know?
- What is the nature of your problem?
- Does the language have many resources such as an Application Programming Interface (API)?
- Are there good compiler which generate fast executable code?

- Is the language platform independent?
- Do you need to control low-level features of the operating system?

I have chosen the C programming language because of the nature of the problem. We need a language that can produce efficient executable code and supplied with a good compiler. I have used the GNU C Compiler with many optimization features. The compiler is available for different operating system such as Unix, Linux and Microsoft Windows. C does not have good support with respect to predefined data structures and algorithms. The programming language Java has quite good support of predefined data structures, but some of the structures used in this implementation are not included. I think that the best solution is to implement the time consuming parts in C, and by using a Java Native Interface (JNI) connect a Java application to the C implementation. The native interface will not be included in this thesis.

4.2 Lexical model

The lexical model contains a list of possible tags for each word and in some cases for each suffix with respect to the training data. A probability is assigned to every possible tag in relation to the word. In chapter 3 I defined different ways to calculate the probability and I also extended the model with suffix probabilities $P(s|t)$. This means that not only words of occurring with different tags have a list of possible tags but also every suffix of a word has a probability. A letter tree can be used to represent the suffix and word tree and the data structure of this representation is called a trie.

The trie data structure is a tree structure often used to represent collections of words and in this application also collections of suffixes. Each node in the trie contains a set of characters which points to the next character in the word. This structure makes it quite fast to look up words in a dictionary since the time to find a word is proportional to the length of the word. The benefit of this data structure is that it is very fast to find, insert and delete words. The major disadvantage is that the trie consumes enormous amounts of memory. If we want to represent every combination in an 8-bit ASCII code the amount of memory will grow exponentially as the tree gets deeper. If every node contains 1028 bytes and the depth of the trie is 3 we need about 17 GBytes of memory. Fortunately we do not have to represent all combinations but only a small subset, because of two things. First of all, the trie will be sparse since most possible strings use not found in the training data. Secondly, we can reduce the character set to a subset of the ASCII codes. To reduce the character set a function is needed to map characters to the data structure in the node that contains links to the next level in the trie. This is done in the module *charset* shown in figure 4.1. For instance, we can map an uppercase letter to the corresponding lowercase letter and let the node separate them. We also have one dummy character to which we map many of ASCII codes which are not human readable and not likely to occur in the training material.

Figure 4.2 shows how the trie node is structured. Each node contains a character, a boolean variable assigned the value true if the sequence of characters ending in this node is a word and an array of pointers to the next level in the trie. The structure also contains two pointers which each points to a linked

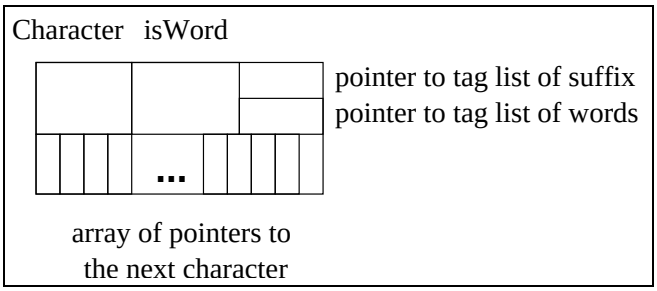

Figure 4.2: Structure of a trie node

list of tags. Each list node contains a tag code, frequencies for capitalized and non-capitalized tokens and a pointer to the next list node.

When creating the trie from the training data we simple create the trie nodes if they do not exist. As mentioned in section 3.2 we insert the word token backwards. When a character is inserted we either increase the frequency in the list node in the suffix list for the particular tag or insert a new list node with frequency one. When we have completed the entire word we update or insert a new list node in the word list according to the tag.

It was quite easy to implement the insert, lookup and delete function. However, save and load functions are more difficult because of the relatively complex structure with trie nodes with pointers to linked lists and an array with pointers to the children. The solution was to do a recursive preorder traversal. First saving the information in the trie node and then the two linked lists of tags. Most of the pointers in the array will not point to a child node because of the sparse structure of the trie. For that reason it is better to save information about how many children the node is pointing at.

4.3 Contextual model

We have seen in previous chapters that the lexical model is not enough to get a good result, we also need a model which takes context into consideration. The contextual model has a window where a tag is dependent on n preceding tags. The variable n defines different n-class models such as the unigram, bigram and trigram model, as discussed in previous chapters.

To implement the contextual model I need n-dimensional arrays to store the frequencies and probabilities for the different models. For the unigram model I use a vector with the size $|T| + 1$ where $|T|$ is the size of the tagset. The vector needs an additional element because of a special start tag which is used in the tagging process. The bigram needs a matrix with the size $(|T| + 1)^2$ and the trigram model with the size $(|T| + 1)^3$. The trigram uses a 2-dimensional matrix representing a 3-dimensional model. A mapping method with modulo calculation maps between the different dimensions. The use of a 2-dimensional matrix simplifies the tagging process because we use a 2-dimensional matrix to represent the trellis in the Viterbi algorithm.

In the training process the data structure is generated by reading tagged text from the corpus and counting the number of occurrences of a tag with $n - 1$ preceding tags. After creating the frequency structures, the probability

structures are calculated according to the model used. The result of the delimiter tag is copied to the first and the special start tag because the segmentation of the text in the tagging process assumes that the model starts with a new sentence, see section 4.4. Finally, the frequency and probability structures are stored in separate files. An internal representation of the tagset is used, each tag being represented by an integer value called tag code. This tag code is used when accessing data in the different structures.

4.4 Segmentation and Viterbi

The previous sections have mostly concerned the implementation of the training process of the contextual and the lexical models. They are of no use if we can not use them to tag a Swedish text and in this section we will discuss the implementation of the tagging process. First the tagger most be initialized with models by loading them from the secondary storage. The next step is to tag an untagged text or as in the empirical evaluation a tagged corpus.

Suppose we have about 100000 untagged words and a tagset with 150 tags and use the trigram model and every probability needs 4 bytes of memory space. This will result in a huge matrix when using the Viterbi algorithm: $4(100000(150+1)^2) \approx 9$ GBytes. We need to segment the untagged text into smaller chunks and then run the Viterbi algorithm on each chunk.

An easy approach is to cut the text into equally sized pieces except for the last piece. Unfortunately this will result in decreased accuracy because cutting can occur in the middle of a sentence and we then lose information about contextual dependencies. A more complex solution is to always cut between two unambiguous words. A disadvantage of this approach is that we need to keep track of unambiguous word or determine it during run-time which is memory and time consuming. If we use the trigram model it will be even more complicated because we need to keep track of three unambiguous words, since the two previous words need to be unambiguous.

My approach is to use sentence delimiters such as period and question mark as markers for a cutting point. Usually there are no contextual dependencies regarding parts-of-speech over the sentence boundary. The segmentation method uses an interval between *cutmin* and *cutmax* where the sentence delimiter can be used for cutting the text. We need to set a maximum size because we need to allocate enough memory for the trellis matrix in advance.

Now we have a mechanism to divide the text into smaller chunks. A chunk refers to a word sequence which will be tagged separately. In section 2.5 we have seen the Viterbi algorithm which helps the tagger to find the most probable path through the Hidden Markov Model.

I have used the following version of the Viterbi algorithm when tagging with a bigram model [9]:

1. Create
    1.1 A path probability matrix $trellis[|T|+1, |S|+1]$
    1.2 A backpointer matrix $backpointer[|T|+1, |S|+1]$
2. For every i, j $trellis[i, j] \leftarrow 0.0$
3. $trellis[0, 0] \leftarrow 1.0$
4. For each word s in sequence S do
    4.1 For each state c in set C do
        4.1.1 For each previous state p in set P do
            4.1.1.1 $newscore \leftarrow trellis[p, s-1] * lexical[s]$
                $*contextual[p, s]$
            4.1.1.2 if $(newscore > trellis[c, s])$ then
                4.1.1.2.1 $trellis[c, s] \leftarrow newscore$
                4.1.1.2.1 $backpointer[c, s] \leftarrow c$
5. $back \leftarrow backpointer[1, |S|]$
6. $max \leftarrow trellis[1, |S|]$
7. For each state t in T do
    7.1 if $(max < trellis[t, |S|]$ then
        7.1.1 $back \leftarrow backpointer[t, |S|]$
8. For each state s from $|S| - 1$ to 1 do
    8.1 Print *back*
    8.2 $back \leftarrow backpointer[back, s]$

Viterbi uses two matrices *trellis* and *backpointer* to complete the task, one for saving the highest probabilities for a specific path and one for saving the actual path. The trellis matrix needs to be initialized with zero probabilities except for the starting state, which is initialized with the highest probability.

To compute the best possible path or tag sequence we need three nested loops. The first loop reviews every symbol s in the input sequence S, which consists of the input sequence of tokens. For every symbol s we have to compute the highest probability for going from a previous state p to the current state c. The set C consists of all possible tags which the lexical model contains for a particular word or suffix and the set P consists of all possible tags for the previous word or suffix. In the innermost loop the algorithm computes a new score by multipling the previous score with the lexical and contextual probabilities. If the new score is higher then the current score, the new score will be assigned as the current score. At the same time the state is saved in the back-pointer matrix to keep track of the path.

When all the columns of the trellis have been filled the algorithm uses the back-pointer matrix to find the best possible tag sequence according to the lexical and contextual model. To find the path through the back-pointer matrix the algorithm takes the corresponding state, which has the highest score in the trellis matrix, and then uses this state to find the tag sequence in the last loop.

In my implementation I actually use logarithms of probabilities to cope with the problem of accuracy of floating-point values, since probabilities may become too small to be represented in the computer. This representation alters the algorithm a little. For example, instead of multipling the probabilities when computing the new score the algorithms sum them together and the starting state is initialized with the value 0.0 and the rest of the matrix initialized with the value 1.0 which indicates an illegal probability.

When the trigram model is used the algorithm usus two preceding tags to calculate the new score and to do so the algorithm uses one more nested loop to review the extra preceding tag.

4.5 Tokenizer

The first step in the analysis of the text input is usually the tokenization of the input into words and punctuation. For many languages such as Swedish this problem is fairly easy in that one can to a first approximation assume that word boundaries consist of white space or punctuation in the input text.

I have implemented an uncomplicated tokenizer which assumes that every token is surrounded by white space, this also including punctuation tokens. In the future I may construct a more sophisticated tokenizer.

This module also includes functions for reading a line from a tokenized and tagged corpus. The tokenizer supports the format for SUC version 1, SUC version 2 and a tab separated word/parts-of-speech corpus. A new tokenizer can easily be implemented in the tokenizer module. When using the module we simple call the *gettoken()* function and the module uses the appropriate tokenizer stated in the option file.

4.6 Tagset

One of the requirements is that the tagger should be corpus independent. This requires an internal representation of the possible tags. The solution is to use an integer to represent a tag. The tagset is stored as a hash table with a structure containing two strings *tag* (part-of-speech) and *m* (morpho-syntatic features). An integer *tagcode* is used in the rest of the program to represent the tags in the lexical and contextual model. Only when the program gets input and generates output are the *tag* and optional *m* used. Optional structure items *word*, and *freq* are not important, but it is quite good to have an example of a word. To create a tagset we either create a file with the following tab-separated format:

```
tag m tagcode word_example freq
```

or use the Tag analyzer program to read a corpus and generate the tagset file. The tagset file also counts the number of occurrences of each tag. This frequency is not used in the training or tagging process, it is only for presentation of the tagset.

4.7 Options

When executing the MALT tagger we need an easy way to vary between different settings. One solution could be to use of a menu where the user choose a between the possible settings. The disadvantage of this approach is that it takes too much time to change the different parameters. Another approach is to invoke the program with different arguments. Usually this approach is good, but the problem is that there are to many options. My solution is to have an option file which the user can manipulate. The option file is loaded when the program starts by entering the path to the option file as an argument:

```
% malttagger -of==/path/filename
```

In the option file we can choose tagset, tokenizer, test methods, contextual model and lexical model. We can also choose between different actions such as training and tagging, or both.

Chapter 5

Evaluation

In this chapter I will present an empirical evaluation of different models for tagging Swedish text. The models are defined in chapter 3 and their implementation is described in chapter 4. I will use a tagged corpus in both the training process and the test phase. In the training process I will use one part of the corpus to create the lexical model by counting the occurrences of a word with a given part-of-speech and the contextual model by counting the occurrences of unigrams, bigrams and trigrams. In the test phase I will use a different part of the corpus to test the accuracy by tagging the tokens and then checking if the tagger has assigned the correct tag.

5.1 Corpus data

The data used for the empirical evaluation come from the Stockholm-Umeå Corpus (SUC) version 2.0. This corpus is a balanced corpus of written Swedish from different areas such as newspaper text and fiction. The version SUC 2.0 is not an official release. The corpus has been automatically tagged for parts-of-speech and then corrected manually [6, 10].

By processing the entire corpus I could identify that the corpus contains $1,166,603$ tokens (punctuation included). The corpus is divided into small files of about 2500 tokens in each file. The tagset used consists of 25 basic parts-of-speech and I will call this tagset the small tagset in the evaluation. The large tagset includes also morpho-syntactic features and the large tagset has 153 distinct tags. The tagsets are described in appendix A and B.

5.2 Evaluation

In the empirical evaluation the Stockholm-Umeå Corpus will be divided in different ways to get as reliable results as possible. First of all I will save a small subset of the corpus to test the tagger when the evaluation of the different models is finished. The final test will only test the best model(s) and the baseline model with respect to their the accuracy rate, giving an estimate of the accuracy when applied to unseen text. In the final test I will also evaluate the part-of-speech tagger's time consumption. I have chosen to use $1,101,483$ tokens to train the models and 65120 tokens to test the tagger in the final test. I

Table 5.1: Cross Validation

Round	**Word types**	**Known words**	**Unknown words**	**Total**
1	86264	103968	8852	112820
2	86804	104758	7784	112542
3	86980	102036	7695	109731
4	86955	102440	7995	110435
5	86487	101443	8682	110125
6	86755	100692	8074	108766
7	86948	100725	8155	108880
8	86752	101213	8172	109385
9	87425	100453	7527	107980
10	86836	102879	7940	110819
Total:		1020607	80876	1101483

randomly selected the files that should be in the training and test set, but with one restriction that there should be a good distribution over different areas of texts.

Before the final test can be done we need to evaluate the different lexical and contextual models and identify the best model(s). Cross-validation is a tool for selection of statistical models and is a very general framework using no special model assumptions. Cross-validation gives an estimate of a model's generalization capabilities [11]. When a cross-validation is performed we divide the data into k subsets of approximately equal size. We then train the model k times, each time leaving out one of the subsets from training. The excluded subset in the training is used to test the model and this will also be performed k times.

I will perform a cross-validation and the training data in the final test will be used for the cross-validation. I have decided to use $k = 10$, which means that the data will be divided into 10 subsets. We can now define a validation round as 9 subsets for training and one subset for testing the tagger. Table 5.1 shows the number of unknown and known words and the total for each testing round. I will not present the accuracy rate for each round, instead the mean will be presented.

When evaluating the different models I will present four different accuracy rates: small tagset with bigram model, small tagset with trigram model, large tagset with bigram model and large tagset with trigram model.

5.3 Results

5.3.1 Lexical model

The lexical model estimates the probability of a word given its part-of-speech $P(w|t)$. In section 3.2 different lexical models are defined. In this section I will perform an evaluation of the models and find out which model is the best according to accuracy. To do the experiment we need to have a contextual model. The contextual models used in the experiment are $P_{CM1}(t_i|t_{i-1})$ and

Table 5.2: Results for the baseline lexical model (LM1)

Tagset/ Contextual model	**Total**	**Known words**	**Unknown words**
Small/Bigram	91.96	95.95	41.69
Small/Trigram	93.60	96.09	62.19
Large/Bigram	89.92	95.39	20.92
Large/Trigram	88.60	93.36	28.53

$P_{CM1}(t_i|t_{i-2}, t_{i-1})$, for bigram and trigram respectively, as defined in section 3.3. No changes to the contextual model will be done under the experiment with the lexical model.

First of all we will test the baseline lexical model, which uses additive smoothing to handle the problem with sparse data. If a token is known in the lexicon the probability $P_{LM1}(w|t)$ is used and if it is unknown $P_{LM1}(w_u|t)$ is used. The results of the experiment are shown in table 5.2.

The results for known words are quite good for the small tagset, but for the unknown words it is really bad. The poor results are more visible for the large tagset, since there are more choices and not enough data to do a good estimation. A more sophisticated smoothing method is needed.

One of the primary purposes of this thesis is to find out if a lexical model with suffix probabilities can be used to improve both tagging accuracy and efficiency. In section 3.2 I have defined five different lexical models with suffix probabilities, summarized below:

- *LM2*: If the token is known we simple use the probability from the baseline model (LM1), otherwise the longest possible suffix s is used to calculate the suffix probability $P_{LS2}(s|t)$.
- *LM3*: The same lexical model as (LM2) but with an important difference. The probability from the baseline is also used when the longest possible suffix belongs to the set of known words with respect to the training data.
- *LM4*: The closed classes are eliminated when the token is unknown.
- *LM5*: When a word occur sentence initially we sum the frequencies for capitalized and non-capitalized occurrences when calculating the probabilities for words and suffixes.
- *LM6*: When a token is unknown will this lexical model use a suffix with maximum length L.

The results for the lexical model (LM2) is presented in table 5.3. We can see a substantial improvement from the baseline lexical model for unknown words and a tiny improvement for the known words. The difference between the small and the large tagset decreases for the unknown words. The lexical models (LM3-LM5) were also tested, but the differences found were very small and surely not statistically significant. The result will not be presented here because the differences between (LM2) and (LM3-LM5) were only a few hundredths of a percentage.

Table 5.3: Results for the lexical model with suffix probabilities (LM2)

Tagset/ Contextual model	**Total**	**Known words**	**Unknown words**
Small/Bigram	95.32	96.16	84.70
Small/Trigram	95.29	96.18	84.13
Large/Bigram	94.43	95.58	79.87
Large/Trigram	92.88	93.97	79.15

Table 5.4: Results for the lexical model with suffix probabilities and maximum suffix length L (LM6)

Tagset/ Contextual model	**4**	**5**	**6**	**7**	**8**	**No limit limit**
Small/Bigram	95.11	95.22	95.30	95.31	95.32	95.32
Small/Trigram	95.16	95.23	95.28	95.29	95.30	95.29
Large/Bigram	94.28	94.37	94.45	94.45	94.45	94.43
Large/Trigram	92.65	92.79	92.88	92.90	92.89	92.88

Another interesting experiment is to vary the maximum length of the suffix, which is defined as the lexical model (LM6). I vary the maximum length of the suffix L from 4 to 8. In table 5.4 we can see the parameter L in the columns and in the rightmost column we have the results from (LM2). To minimize the numbers I have excluded the data about known and unknown words. We can see a slight improvement with a maximum suffix length around 7, but it is not a significant improvement. The lexical model (LM6) needs at least to have a maximum suffix length greater than 4.

To summarize the results of the lexical model we can see a difference between the baseline lexical model and the lexical model with suffix probabilities, which is found to be statistically significant given clear the size of the datasets. We can not find any difference between the lexical models with suffix probabilities. Based on the evaluation of the lexical model I have decided to use the lexical model (LM2) for the forthcoming test, since there is no visible difference between the lexical models (LM2-LM6), and the lexical model (LM2) is the most straightforward way to implement a lexical model with suffix probabilities.

5.3.2 Contextual model

In the previous chapters we have concluded based on previous work in this area that a lexical model is not enough to find the tag sequence for a given string of words. We also need information about the context where the word is found. The contextual model is created by computing the probability of a part-of-speech conditioned on a sequence of previous tags. In section 3.3 I decided to use a bigram model $P(t_i|t_{i-1})$ and a trigram model $P(t_i|t_{i-2}, t_{i-1})$. These probabilities can be computed in different ways such as with additive smoothing (CM1), with linear interpolation with relative frequencies (CM2) and with linear interpolation with additive smoothing(CM3). In this evaluation the lexical model will be the same as in section 5.3.1 I decided to use the lexical

Table 5.5: Results for the contextual model with linear interpolation of relative frequencies (CM2)

Tagset/ Contextual model	**Total**	**Known words**	**Unknown words**
Small/Bigram	95.32	96.17	84.67
Small/Trigram	95.33	96.18	84.59
Large/Bigram	94.49	95.65	79.98
Large/Trigram	85.45	86.57	71.29

Table 5.6: Results for the contextual model with linear interpolation of additive smoothing (CM3)

Tagset/ Contextual model	**Total**	**Known words**	**Unknown words**
Small/Bigram	95.33	96.17	84.70
Small/Trigram	95.59	96.45	84.68
Large/Bigram	94.44	95.60	79.90
Large/Trigram	94.77	95.91	80.34

model (LM2).

One of the easiest way to calculate the probability is to use relative frequencies without any smoothing methods, but this is not a good idea because of the sparse data problem. With the small tagset and bigram model maybe there will not be any zero-frequencies and maybe the result is not too bad, but with the trigram model there will be many zero-frequencies. A contextual model which only uses on relative frequencies is not reliable and will not be used.

To eliminate the problem with zero-frequencies we must use a smoothing method. In tables 5.2 and 5.3 I used additive smoothing for the contextual models (CM1). The purpose of using the more complex trigram model is to have better accuracy rate than the bigram model, but still the results are quite poor. Due to the lack of training data the trigrams in some cases can not be estimated properly [15].

Another smoothing method is to experiment with a linear interpolation of the unigram, the bigram and the trigram. Linear interpolation can maybe improve the accuracy rates specially for the trigram model. The experiment with linear interpolation will be done with both relative frequencies (CM2) and additive smoothing (CM3), and the results are shown in table 5.5 and 5.6.

In an ad hoc experiment with different optimization parameters λ_1, λ_2 and λ_3 I found that the best accuracy rate is about $\lambda_1 = 0.02$ and $\lambda_2 = 0.98$ for the bigram model and $\lambda_1 = 0.14$, $\lambda_2 = 0.53$ and $\lambda_3 = 0.33$ for the trigram model. The attentive reader notes that bigram model (CM2, CM3) practically does not use any unigram in the linear interpolation. The improvement of the bigram model (CM2, CM3) is negligeable. The improvement of the trigram model with the contextual model using linear interpolation of additively smoothed n-grams (CM3) is considerable better.

The evaluation of the contextual model shows that the model with linear interpolation as smoothing method should be used for the trigram model. How-

Table 5.7: Final test result of the baseline model (LM1,CM1)

Tagset/ Contextual model	**Total**	**Known words**	**Unknown words**	**Time (s)**
Small/Bigram	91.14	95.67	39.42	0.5
Small/Trigram	93.03	95.80	61.80	0.7
Large/Bigram	89.23	95.17	21.42	2.4
Large/Trigram	87.98	93.19	28.60	5.5

Table 5.8: Final test result of the chosen model (LM2,CM3)

Tagset/ Contextual model	**Total**	**Known words**	**Unknown words**	**Time (s)**
Small/Bigram	94.92	95.95	83.12	0.4
Small/Trigram	95.12	96.23	83.40	0.6
Large/Bigram	94.13	95.46	78.94	0.5
Large/Trigram	94.43	95.74	79.44	0.9

ever, the bigram model can be smoothed with both additive and linear interpolation. The results also show that the trigram model is slightly better than the bigram model. For the final test I choose the contextual model with linear interpolation based on additive smoothing (CM3).

5.3.3 Final test

Before performing the cross-validation I saved a small subset of the Stockholm-Umeå Corpus to do the final test. Only using the result from the cross-validation is not reliable because optimization parameters have been changed to get the best possible result for these data. In the final test I will only evaluate the chosen lexical model with suffix probabilities (LM1) and contextual model (CM3). I will also conduct a final test on the baseline model (LM1 and CM2) to have something to compare with.

The time spent tagging the words will also be measured. The presented time-period will only contain the tagging process, not loading the lexical and contextual model from file and initializing and deleting the data structures used. I will perform the test three times for each experiment and then present the average of the time measurement. In the final test I will use 1101483 tokens for the training process and 65120 tokens for the test. The training data contains 93024 distinct word types. The test data consist of 59872 known words and 5248 unknown words in relation to the training data. The final test will be conducted on a Sun Enterprise 450 with four 480 MHz UltraSPARCII processors and with 4 GByte memory, only one processor will be used in the test since the program is not parallelized. Table 5.7 presents the results of the final test for the baseline model and table 5.8 presents the results for the chosen model. Both the baseline model and the chosen model have a lower accuracy rate compared to the cross-validation for the same models. In all the experiments we can see that the chosen model has substantially better accuracy rate than the baseline model. The greatest gap between the models occur with the trigram model and

the large tagset and this is quite natural due to the sparse data problem in this model. The time measurements show that the chosen model have a considerably better performance than the baseline, especially when the large tagset is used.

5.4 Discussion

The main result concerning the choice of lexical model is that the lexical model with suffix probabilities outperforms the baseline model with respect to the data at hand. If we want to use a lexical model without suffix probabilities we need a more sophisticated smoothing model. However, with the lexical model with suffix probabilities we eliminate the need for a sophisticated smoothing model. It seems to be enough with additive smoothing to obtain good accuracy. Moreover, in the baseline model we need to keep track of the open classes of parts-of-speech, which increases the number of states to be calculated in the tagging process. In one of the experiments with the lexical model with suffix probabilities I used a closed class elimination method, but the accuracy rate was not improved. Thus, a lexical model with suffix probabilities seems to eliminate the need for such a method as well. Another interesting result was that the special treatment of tokens occurring at the start of a sentence did not improve the performance of the tagger. Maybe a more complex method could improve the accuracy rate by taking into account other dependencies at the start of a sentence, but it is hard to define a model which does not decrease the accuracy for other words. My results almost confirm the results which Carlberger & Kann [3] got concerning the maximum suffix length. They did not get any improvement with a maximum suffix length of more than 5 letters. In my experiment I did get a slight improvement with 6 letters, but the increase in accuracy rate was not significantly better than for 5 letters.

Comparing my results with others is rather difficult, because of different training data, tagsets and empirical methods. Nivre [15] has shown that Good-Turing estimation for the lexical model outperforms the additive smoothing especially for unknown words. However, with lexical additive smoothing in combination with a lexical model with suffix probabilities the results are better than the results obtained by Nivre (small tagset 94.82% and large tagset 91.46%) [15]. Especially for the large tagset the result 94.43% is substantially better. The *GRANSKA* tagger created by Carlberger & Kann uses 140 tags, incorporates *Svenska Akademiens ordlista* (SAOL) and in some cases consider a token as consisting of two or more words. They reports a tagging accuracy of 96.4% [3].

One question we can ask ourselves, is whether the results would be even better with a combination of Good-Turing and a suffix model, but this experiment is beyond this thesis. Probably the answer is that the results would be slightly improved, but I do not think that any major difference will be detected.

Concerning the contextual model, we have seen that the linear interpolation based on additive smoothing of n-grams outperforms the simple additive smoothing when we have a problem with sparse data, which is the case in the trigram model especially with a large tagset. Nivre [15] has shown that with respect to the data at hand both additive smoothing and back-off smoothing is better than Good-Turing estimation. My results indicate that linear interpolation based on additive smoothing of n-grams outperforms additive smoothing,

hence also the other methods.

A great advantage of using a lexical model with suffix probabilities is that a letter tree can be used to store both suffixes and words which speeds up the tagging. When an unknown word occurs we simply use the suffix probabilities, which means that the lookup of unknown words is fast (in fact faster than if the word had been known). Moreover the use of suffixes reduces the number of possible tags for each word, which also improves efficiency. The performance regarding time consumption improves dramatically when tagging with the large tagset and the trigram contextual model compared to the baseline model, which has to consider every open class as a candidate tag for an unknown word. The time difference when tagging around 65000 words with the small and large tagset is very small with the bigram model. With the trigram model the difference is slightly larger but still surprisingly low.

In section 2.5 we noticed that the Viterbi algorithm is linear in the number of words to be tagged, but quadratic in the number of states. The suffix model reduces the number of states considered compared to the baseline model which speeds up the tagger. In addition, it makes lookup of lexical probabilities very fast.

Chapter 6

Conclusion

The most important conclusion from this master's thesis is that the lexical model with suffix probabilities improves the tagging of Swedish text in two different ways. First, it improves the tagging of unknown words dramatically compared to a simple additive smoothing, and a slight improvement can also be detected when comparing the result with a more sophisticated smoothing method such as Good-Turing. Secondly, it improves efficiency since suffixes and words are stored in same structure which speeds up lexical lookup. The time complexity of lexical lookup is linear according to the length of the word or suffix. The model also speeds up the process of tagging, since the number of states considered when finding the optimal path through the Hidden-Markov model decreases.

Using suffix analysis to improve the tagging of unknown words has been used in several previous studies [16, 3]. However, in this thesis I have shown that integrating the word and suffix model using an appropriate data structure not only increases accuracy but also efficiency.

The goal of this master's thesis was to define tagging models and implement these models in such a way that the tagger could accomplish an accuracy rate around 95 percent for the Stockholm-Umeå corpus with both the small tagset and the large tagset. Although the final results with the large tagset are slightly below 95 percent the results indicate that this goal can be achieved with a lexical model using suffix probabilities and a contextual model with linear interpolation based on additive smoothing of n-grams. Another goal was to accomplish this in an efficient manner, meaning that the tagger should tag 20000 tokens in a second. In the end, the tagger tags around 70000 tokens per second on a Sun Enterprise 450 with four 480 MHz UltraSPARCII processors and with 4 GByte memory, using only one processor in the test. Although it is hard to compare actual running time in a fair manner, since the test conditions are different from time to time and from machine to machine, we may compare this with Carlberger & Kann [3] who reports a tagging rate of 14000 words per seconds on a Sun Sparcstation 10.

The MALT tagger cannot yet be considered as a professional part-of-speech tagger. First of all, the tagger needs a better tokenizer, either by incorporating an existing tokenizer or by implementing more sophisticated one. To improve the tagging accuracy we need to do more experiments with both the lexical and the contextual model. Another important aspect is to develop an interface that other applications can use to communicate with the tagger.

Bibliography

[1] A. V. Aho and J. D. Ullman. *Foundations of Computer Science*. Computer Science Press, 1992.

[2] E. Brill. Part-of-speech tagging. In R. Dale, H. Moisl, and H. Somers, editors, *Handbook of Natural Language Processing*, pages 403–414. Marcel Dekker, 2000.

[3] J. Carlberger and V. Kann. Implementing an efficient part-of-speech tagger. *Software — Practice and Experience*, 29(9):815–832, 1999.

[4] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics*, 1996.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The Massachusetts Institute of Technology, 1990.

[6] E. Ejerhed, G. Källgren, O. Wennstedt, and M. Åström. The linguistic annotation system of the Stockholm-Umeå corpus project. Technical Report 33, University of Umeå: Department of Linguistics, 1992.

[7] W.A. Gale and G. Sampson. Good-Turing frequency estimation without tears. *Journal of Quantitative Linguistics*, 2:217–237, 1995.

[8] L. Grönqvist and M. Gunnarsson. A method for finding word clusters in spoken language. In *Proceedings for Corpus Linguistics 2003 conference*. UCREL, Lancaster, March 2003.

[9] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice-Hall, 2000.

[10] G. Källgren. Documentation of the Stockholm Umeå Corpus. Technical report, University of Umeå: Department of Linguistics, 1998.

[11] F. Leisch, K. Hornik, and L. C. Jain. Nn classifiers: Reducing the computational cost of cross-validation by active pattern selection. In *2nd New Zealand Two-Stream International Conference on Artificial Neural Networks and Expert Systems (ANNES '95)*, pages 91–95, 1995.

[12] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[13] T. McEnery and A. Wilson. *Corpus Linguistics*. Edinburgh University Press, 1996.

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

[15] J. Nivre. Sparse data and smoothing in statistical part-of-speech tagging. *Journal of Quantitative Linguistic*, 7:1–17, 2000.

[16] C. Samuelsson. Morphological tagging based entirely on Bayesian inference. In *NODALIDA'93. Proceeding of the '9:e Nordiska Datalingvistikdagarna'*, pages 225–238, 1994.

[17] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory*, 13:260–269, 1967.

Appendix A

The SUC small tagset

Tag	Part-of-Speech	Tag code	Example
AB	Adverb	7	inte
DT	Determiner	9	en
HA	Wh adverb	13	när
HD	Wh determiner	24	vilka
HP	Wh pronoun	15	vad
HS	Wh possessive	25	vars
IE	Infinitive marker	19	att
IN	Interjection	23	ja
JJ	Adjective	8	mycket
KN	Conjunction	5	och
MAD	Delimiter	4	.
MID	Delimiter	10	-
NN	Noun	1	Smygrustning
PAD	Delimiter	18	”
PC	Participle	12	ansedda
PL	Particle	17	under
PM	Proper name	3	MATS
PN	Pronoun	11	en
PP	Preposition	2	av
PS	Possessive	20	sina
RG	Cardinal numeral	21	två
RO	Ordinal numberal	22	första
SN	Subjunction	14	att
UO	Foreign word	16	thinktank
VB	Verb	6	har

Appendix B

The SUC large tagset

This table is taken from the documentation of the Stockholm Umeå Corpus [10].

Feature	**Value**	**Legend**	**Parts-of-speech where feature applies**
Gender	UTR	Uter (common)	DT, HD, HP, JJ, NN, PC, PN, PS, (RG, RO)
	NEU	Neuter	
	MAS	Masculine	
Number	SIN	Singular	DT, HD, HP, JJ, NN, PC, PN, PS, (RG, RO)
	PLU	Plural	
Definiteness	IND	Indefinite	DT, (HD, HP, HS), JJ, NN, PC, PN, (PS, RG, RO)
	DEF	Definite	
Case	NOM	Nominative	JJ, NN, PC, PM, (RG, RO)
	GEN	Genitive	
Tense	PRS	Present	VB
	PRT	Preterite	
	SUP	Supinum	
	INF	Infinite	
Voice	AKT	Active	
	SFO	S-form	
Mood	KON	Subjunctive	
Participle form	PRS	Present	PC
	PRF	Perfect	
Degree	POS	Positive	(AB), JJ
	KOM	Comparative	
	SUV	Superlative	
Pronoun form	SUB	Subject form	PN
	OBJ	Object form	
	SMS	Compound	All parts-of-speech

Tag	Tag code	Example
AB	11	inte
AB AN	90	kl
AB KOM	29	oftare
AB POS	31	fullkomligt
AB SMS	124	upp-
AB SUV	53	främst
DT AN	153	d
DT MAS SIN DEF	130	Denne
DT MAS SIN IND	129	samme
DT NEU SIN DEF	22	det
DT NEU SIN IND	56	ett
DT NEU SIN IND/DEF	106	allt
DT UTR SIN DEF	41	den
DT UTR SIN IND	16	en
DT UTR SIN IND/DEF	110	all
DT UTR/NEU PLU DEF	20	de
DT UTR/NEU PLU IND	85	Några
DT UTR/NEU PLU IND/DEF	66	alla
DT UTR/NEU SIN DEF	131	vardera
DT UTR/NEU SIN IND	70	Varje
DT UTR/NEU SIN/PLU IND	75	samma
HA	25	när
HD NEU SIN IND	125	Vilket
HD UTR SIN IND	105	vilken
HD UTR/NEU PLU IND	104	vilka
HP - - -	36	som
HP NEU SIN IND	35	vad
HP NEU SIN IND SMS	149	vad-
HP UTR SIN IND	108	Vem
HP UTR/NEU PLU IND	92	vilka
HS DEF	107	vars
IE	48	att
IN	103	ja
JJ AN	111	tf
JJ KOM UTR/NEU SIN/PLU IND/DEF GEN	127	svagares
JJ KOM UTR/NEU SIN/PLU IND/DEF NOM	50	senare
JJ KOM UTR/NEU SIN/PLU IND/DEF SMS	140	äldre-
JJ POS MAS SIN DEF GEN	138	intellektuelles
JJ POS MAS SIN DEF NOM	67	irakiske
JJ POS NEU SIN IND GEN	151	angelägets
JJ POS NEU SIN IND NOM	12	mycket
JJ POS NEU SIN IND/DEF NOM	117	eget
JJ POS UTR - - SMS	123	låg-
JJ POS UTR SIN IND GEN	119	anhörigs
JJ POS UTR SIN IND NOM	24	optimistisk
JJ POS UTR SIN IND/DEF NOM	82	egen
JJ POS UTR/NEU - - SMS	132	utrikes-

Tag	Tag code	Example
JJ POS UTR/NEU PLU IND NOM	84	flera
JJ POS UTR/NEU PLU IND/DEF GEN	99	dödas
JJ POS UTR/NEU PLU IND/DEF NOM	21	politiska
JJ POS UTR/NEU SIN DEF GEN	134	flirtigas
JJ POS UTR/NEU SIN DEF NOM	13	Gamla
JJ POS UTR/NEU SIN/PLU IND NOM	133	rätt
JJ POS UTR/NEU SIN/PLU IND/DEF NOM	63	långtgående
JJ SUV MAS SIN DEF GEN	152	äldstes
JJ SUV MAS SIN DEF NOM	122	siste
JJ SUV UTR/NEU PLU DEF NOM	93	flesta
JJ SUV UTR/NEU PLU IND NOM	139	Flest
JJ SUV UTR/NEU SIN/PLU DEF NOM	47	bästa
JJ SUV UTR/NEU SIN/PLU IND NOM	109	starkast
KN	9	och
KN AN	120	&
MAD	6	.
MID	18	-
NN - - - -	57	fjol
NN - - - SMS	142	styr-
NN AN	95	kr
NN NEU - - -	141	orda
NN NEU - - SMS	76	hotell-
NN NEU PLU DEF GEN	94	Landstingens
NN NEU PLU DEF NOM	8	stormaktsblocken
NN NEU PLU IND GEN	89	års
NN NEU PLU IND NOM	3	raketvapen
NN NEU SIN DEF GEN	44	områdets
NN NEU SIN DEF NOM	14	testamentet
NN NEU SIN IND GEN	102	buds
NN NEU SIN IND NOM	28	töväder
NN UTR - - -	114	dags
NN UTR - - SMS	96	planerings-
NN UTR PLU DEF GEN	62	irakiernas
NN UTR PLU DEF NOM	17	konflikterna
NN UTR PLU IND GEN	74	nationers
NN UTR PLU IND NOM	27	missiler
NN UTR SIN DEF GEN	60	västvärldens
NN UTR SIN DEF NOM	7	Avspänningen
NN UTR SIN IND GEN	88	fredags
NN UTR SIN IND NOM	1	Smygrustning
PAD	43	”
PC AN	116	tf
PC PRF MAS SIN DEF GEN	113	knivhuggnes
PC PRF MAS SIN DEF NOM	101	utsände
PC PRF NEU SIN IND NOM	71	känt
PC PRF UTR SIN IND GEN	145	uppsagds
PC PRF UTR SIN IND NOM	38	ansedd
PC PRF UTR/NEU PLU IND/DEF GEN	81	deporterades

Tag	Tag code	Example
PC PRF UTR/NEU PLU IND/DEF NOM	73	kallade
PC PRF UTR/NEU SIN DEF GEN	135	åtalades
PC PRF UTR/NEU SIN DEF NOM	23	ansedda
PC PRS UTR/NEU SIN/PLU IND/DEF GEN	143	asylsökandes
PC PRS UTR/NEU SIN/PLU IND/DEF NOM	39	oberoende
PL	42	under
PM GEN	5	DN:s
PM NOM	4	MATS
PM SMS	118	Göteborgs-
PN MAS SIN DEF SUB/OBJ	86	denne
PN NEU SIN DEF SUB/OBJ	32	det
PN NEU SIN IND SUB/OBJ	72	någonting
PN UTR PLU DEF OBJ	91	oss
PN UTR PLU DEF SUB	61	Vi
PN UTR SIN DEF OBJ	33	mig
PN UTR SIN DEF SUB	26	han
PN UTR SIN DEF SUB/OBJ	65	den
PN UTR SIN IND SUB	55	man
PN UTR SIN IND SUB/OBJ	19	en
PN UTR/NEU PLU DEF OBJ	78	dem
PN UTR/NEU PLU DEF SUB	37	de
PN UTR/NEU PLU DEF SUB/OBJ	97	dessa
PN UTR/NEU PLU IND SUB/OBJ	79	många
PN UTR/NEU SIN/PLU DEF OBJ	68	sig
PP	2	av
PP AN	144	inkl.
PS AN	148	h:s
PS NEU SIN DEF	77	sitt
PS UTR SIN DEF	64	sin
PS UTR/NEU PLU DEF	51	sina
PS UTR/NEU SIN/PLU DEF	83	deras
RG GEN	100	fyras
RG NEU SIN IND NOM	98	ett
RG NOM	58	två
RG SMS	126	1950-
RG UTR SIN IND NOM	80	en
RO GEN	147	III:s
RO MAS SIN IND/DEF GEN	112	andres
RO MAS SIN IND/DEF NOM	121	andre
RO NOM	59	första
RO SMS	146	första-
SN	34	att
UO	40	thinktank
VB AN	136	jfr
VB IMP AKT	69	låt
VB IMP SFO	128	minns
VB INF AKT	15	ge
VB INF SFO	54	regleras

Tag	Tag code	Example
VB KON PRS AKT	115	vare
VB KON PRT AKT	87	vore
VB KON PRT SFO	150	funnes
VB PRS AKT	10	har
VB PRS SFO	52	betingas
VB PRT AKT	49	hänvisade
VB PRT SFO	46	tillfrågades
VB SMS	137	läs-
VB SUP AKT	45	myntat
VB SUP SFO	30	synts