KTH Numerical Analysis
and Computer Science

Automatic and Unsupervised Methods in Natural Language Processing

JOHNNY BIGERT

Doctoral Thesis
Stockholm, Sweden 2005

TRITA-NA-0508
ISSN 0348-2952
ISRN KTH/NA/R-05/08-SE
ISBN 91-7283-982-1

KTH Numerisk analys och datalogi
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen fredagen den 8 april 2005 i Kollegiesalen, Administrationsbyggnaden, Kungl Tekniska högskolan, Valhallavägen 79, Stockholm.

Tryck: Universitetsservice US AB

Abstract

Natural language processing (NLP) means the computer-aided processing of language produced by a human. But human language is inherently irregular and the most reliable results are obtained when a human is involved in at least some part of the processing. However, manual work is time-consuming and expensive. This thesis focuses on what can be accomplished in NLP when manual work is kept to a minimum.

We describe the construction of two tools that greatly simplify the implementation of automatic evaluation. They are used to implement several supervised, semi-supervised and unsupervised evaluations by introducing artificial spelling errors. We also describe the design of a rule-based shallow parser for Swedish called GTA and a detection algorithm for context-sensitive spelling errors based on semi-supervised learning, called ProbCheck.

In the second part of the thesis, we first implement a supervised evaluation scheme that uses an error-free treebank to determine the robustness of a parser when faced with noisy input such as spelling errors. We evaluate the GTA parser and determine the robustness of the individual components of the parser as well as the robustness for different phrase types. Second, we create an unsupervised evaluation procedure for parser robustness. The procedure allows us to evaluate the robustness of parsers using different parser formalisms on the same text and compare their performance. Five parsers and one tagger are evaluated. For four of these, we have access to annotated material and can verify the estimations given by the unsupervised evaluation procedure. The results turned out to be very accurate with few exceptions and thus, we can reliably establish the robustness of an NLP system without any need of manual work.

Third, we implement an unsupervised evaluation scheme for spell checkers. Using this, we perform a very detailed analysis of three spell checkers for Swedish. Last, we evaluate the ProbCheck algorithm. Two methods are included for comparison: a full parser and a method using tagger transition probabilities. The algorithm obtains results superior to the comparison methods. The algorithm is also evaluated on authentic data in combination with a grammar and spell checker.

Sammanfattning

Datorbaserad språkbehandling (natural language processing, NLP) betyder som ordet antyder behandling av mänskligt språk med datorns hjälp. Dock är mänskligt språk väldigt oregelbundet och de bästa resultaten uppnås när man tar en människa till hjälp vid behandlingen av text. Tyvärr är manuellt arbete tidskrävande och därför kostsamt. Denna avhandling fokuserar på vad som kan åstadkommas i NLP när andelen manuellt arbete hålls till ett minimum.

Först beskrivs designen av två verktyg som underlättar konstruktion av automatisk utvärdering. De introducerar artificiella stavfel i text för att skapa övervakad, delvis övervakad och oövervakad utvärdering (supervised, semi-supervised och unsupervised, resp.). Jag beskriver också en regelbaserad parser för svenska vid namn GTA och en detektionsalgoritm för kontextkänsliga stavfel baserad på delvis övervakad inlärning vid namn ProbGranska (ProbCheck).

I den andra delen av avhandlingen skapar jag först en övervakad utvärdering som använder sig av en felfri trädbank för att fastställa robustheten hos parserns komponenter och robustheten för olika frastyper. Därefter skapas en oövervakad utvärderingsmetod för robusthet hos parsrar. I och med detta kan man utvärdera parsrar som använder olika parserformalismer på samma text och jämföra deras prestanda. Fem parsrar och en taggare deltog i utvärderingen. För fyra av dessa fanns ett facit och man kunde bekräfta att uppskattningarna som erhållits från den oövervakade utvärderingen var tillförlitliga. Resultaten visade sig vara mycket bra med få undantag. Man kan därför med god noggrannhet uppskatta robustheten hos ett NLP-system utan att använda manuellt arbete.

Jag utformar därefter en oövervakad utvärdering för stavningsprogram. Med hjälp av denna genomförs en mycket detaljerad analys av tre stavningsprogram för svenska. Sist utvärderas ProbGranska. Jag använder två metoder för jämförelse: en fullparser och en metod som använder övergångssannolikheter hos taggare. Slutsatsen blir att ProbGranska får bättre resultat än båda jämförelsemetoderna. Dessutom utvärderas ProbGranska på autentiska data tillsammans med en grammatikgranskare och ett stavningsprogram.

Acknowledgments

During my time at Nada, I have met a lot of inspiring people and seen many interesting places. I wish to thank

- Viggo for having me as a PhD student and supervising me
- Ola, Jonas S for excellent work
- Joel, Jonas H and Mårten for $Jo^3 \cdot M$
- Jesper for PT, Gustav for poker, Anna, Douglas, Klas, Karim
- Johan, Stefan A, Mikael
- Stefan N, Lars, Linda, Jakob
- Magnus, Martin, Hercules, Johan, Rickard for NLP
- The rest of the theory group at Nada
- Nada for financing my PhD
- Mange for McBrazil and panic music
- Anette, Fredrik, German girl, French guy and others for fun conferences
- Joakim Nivre, Jens Nilsson, Bea for providing their parsers
- Tobbe for keeping in touch, Stefan Ask for lumpen, Lingonord
- Niclas, Rickard, Conny for being good friends
- My family and especially my father for proof-reading

Last, I wish to thank Anna for her support and understanding.

Contents

Chapter 1

Introduction

Personal computers were introduced in the early 70's and today, almost everybody have access to a computer. Unfortunately, the word 'personal' does not refer to the social skills of the machine, nor the fact that the interaction with a computer is very personal. On the contrary, the interaction with a modern computer is via a keyboard, a mouse and a screen and does not at all resemble the way people communicate.

Evidently, spoken language is a more efficient means of communication than using a keyboard. Movies illustrating the future have also adopted this view, as many future inhabitants of earth speak to their computer instead of using a keyboard (e.g. Star Trek Voyager, 1995–2001). The use of a computer to react to human language is an example of Natural Language Processing (NLP). Albeit, spoken interfaces are not very wide-spread.

Another, more widespread application of NLP is included in modern word processors. You input your text and the computer program will point out to you the putative spelling errors. It may also help you with your grammar. For example, if you write 'they was happy', your word processing program would most certainly tell you that this is not correct.

For a grammar checker to be successful, it needs to know the grammar of the language to be scrutinized. This grammar can be obtained from e.g. a book on grammar, in which a human has collected the grammar. Another approach would be to have a computer program construct the grammar automatically from a text labeled with grammatical information. Both approaches have their pros and cons. For example, structuring a grammar manually gives a relatively accurate result but is very time-consuming and expensive, while the computer generation of a grammar is portable to other languages but may not be as accurate.

Clearly, automation is very valuable in all parts of NLP if good enough accuracy can be achieved. Automatic methods are cheap, fast, and consistent and can be easily adapted for other languages, domains and levels of detail. This thesis addresses the topic of automated processing of natural language, or more specific-

ally, two different types of automation. The first was mentioned above, where a computer program automatically gathers data from a *corpus*, which is a large text containing extra, manually added information. This is called a *supervised* method. The second type of automation is where a computer program operates on raw text without extra information. This is called an *unsupervised* method.

1.1 Grammatical and Spelling Errors

To illustrate the use of NLP in everyday life, we use a grammar checker as an example. Checking the grammar of a sentence involves several techniques from the NLP research area. First, the program has to identify the words of the sentence. This is easy enough in languages that use spaces to separate the words, whereas other languages, such as written Chinese, do not have any separation between words. There, all characters of a sentence are given without indication of word boundaries and one or more characters will constitute a word. Thus, a trivial task in one language may be difficult in another.

The second task for a grammar checker is often to assign a *part-of-speech* (PoS) label to each word. For example, in the sentence 'I know her', the first and the third words are pronouns and the second word is a verb. PoS information often include a morphological categorization. To be able to analyze our earlier example 'They was happy', we need to know that 'They' is a plural word while 'was' is singular. Hence, a grammar checking program operating on these facts will realize that a pronoun in plural is inconsistent with a verb in singular. The PoS and morphological information for a word constitute what is called a PoS *tag*.

Assigning PoS tags to an unambiguous sentence is easy enough. The problem arises when a word has more than one possible PoS category, as in the sentence 'I saw a man'. The word 'saw' could either be a verb or a noun. As a human, we realize that 'saw' is a verb that stems from 'see' or the sentence would make no sense. A computer, on the other hand, has no notion of the interpretation of a sentence and thus, it has to resort to other means. Another difficulty in determining the PoS tag of a word is the occurrence of unknown words. For these, we have to make a qualified guess based on the word itself and the surrounding words.

Several techniques have been proposed to assign PoS tags to words. Most tagging techniques are based on supervised learning from a corpus containing text with additional PoS tag information. From the data gathered from the corpus, we can apply several different approaches. One of the most successful is using the data to construct a second-order hidden Markov model (HMM). A widespread implementation of an HMM tagger is Tags'n'Trigrams (TnT) (Brants, 2000). Other techniques for PoS tagging using supervised learning are transformation-based learning (Brill, 1992), maximum-entropy (Ratnaparkhi, 1996), decision trees (Schmid, 1994) and memory-based learning (Daelemans et al., 2001). Hence, a PoS tagger is an excellent example of a supervised method since it requires no manual

work (provided a corpus) and is easily portable to other languages and PoS tag types.

The order between the words of a sentence is not randomly chosen. Adjacent words often form groups acting as one unit (see e.g. Radford, 1988). For example, the sentence 'the very old man walked his dog' can be rearranged to 'the dog was walked by the very old man'. We see that 'the very old man' acts as one unit. This unit is called a *constituent*. Determining the relation between words is called *parsing*. Using the example from above, 'the very old man walked his dog' can be parsed as follows: '[S [NP the [AP very old] man] [VP walked] [NP his dog]]', where S means sentence (or start), NP means noun phrase, AP means an adjective phrase and VP is a verb phrase. Note here that the AP is inside the first NP. In fact, the AP 'very old' could be further analyzed since 'very' is by itself an adverbial phrase. If all words are subordinated a top node (S), we have constructed a *parse tree* and we call this *full parsing*. As a complement to full parsing, we have a technique called *shallow parsing* (Abney, 1991; Ramshaw and Marcus, 1995; Argamon et al., 1998; Munoz et al., 1999). There, we do not construct a full parse tree, but only identify major constituents. Removing the outmost bracket (S) would result in a shallow parse of the sentence. Another level of parse information is *chunking*, where only the largest constituents are identified and their interior is left without analysis (see e.g. the CoNLL chunking competition, Tjong Kim Sang and Buchholz, 2000). Thus, chunking the above sentence would give us '[NP the very old man] [VP walked] [NP his dog]'. Chapter 5 is devoted to the implementation of a rule-based shallow parser for Swedish, capable of both phrase constituency analysis and phrase transformations.

The phrase constituency structure is often described by a Context-Free Grammar (CFG). The CFG formalism actually dates back to the 1950's from two independent sources (Chomsky, 1956; Backus, 1959). Hence, the idea of describing natural language using formal languages is not at all new.

Another widespread type of parse information is given by *dependency grammars*, also originating from the 1950's (Tesnière, 1959). Here, the objective is to assign a relation between pairs of words. For example, in the sentence 'I gave him my address' (from Karlsson et al., 1995; Järvinen and Tapanainen, 1997), 'gave' is the main word having a subject 'I', an indirect object 'him' and a direct object 'address'. Furthermore, 'address' has an attribute 'my'.

Given the phrase constituents of the sentence, we can now devise a grammar checker. As a first example, we check the agreement between words inside a constituent. For example, the Swedish sentence 'jag ser ett liten hus' (I see a little house) contains a noun phrase 'ett liten hus' (a little house). Swedish grammar dictates that inside the noun phrase, the gender of the adjective must agree with the gender of the noun. In this case, the gender of 'liten' (little) is non-neuter while the gender of 'hus' (house) is neuter. Thus, the grammar checker has detected an inconsistency. To propose a correction, we change the adjective to neuter, giving us 'ett litet hus' (a little house).

When the morphological features within the phrases agree, we turn to the overall

agreement of the sentence constituents. For example, in the sentence 'the parts of the widget was assembled', we violate the agreement between the noun phrase 'the parts of the widget' and the verb 'was'. A first step to detect this discrepancy is to determine the number of the noun phrase. To this end, we note that the head of the noun phrase is 'the parts' and thus, it is plural. Now, the number of the noun phrase can be compared to the number of the verb. Clearly, there are many different ways to construct a noun phrase (not to mention other phrase types), which will require a comprehensive grammar to cover them all.

See Section 2.2 for a short description of an implementation of a grammar checker called GRANSKA. The GRANSKA framework was also used for the implementation of the shallow parser in Chapter 5.

Context-sensitive Spelling Errors

Full parsing is a difficult task. Writing a grammar with reasonable coverage of the language is time-consuming and may never be perfectly accurate. Instead, many applications use shallow parsing to analyze the text. Since shallow parsers may leave parts of the sentence without analysis, we do not know whether this is because the text does not belong to a phrase or the fact that the sentence is ungrammatical. Even with at full parser, we cannot determine whether a part of a sentence is left without analysis due to limitations in the grammar or due to ungrammaticality. Using a grammar checker, we can construct rules for many common situations where human writers produce ungrammatical text. On the other hand, since it is very difficult to produce a perfect grammar for the language, we will not be able to construct grammar-checking rules for all cases. For example, spelling errors can cause difficult sentences to analyze as in 'I want there apples'. All of the words in this sentence are present in the dictionary. Nevertheless, given the context, the word 'there' is probably misspelled since the sentence does not have a straightforward interpretation. We see that the correct word could be either 'three' (a typographical error) or 'their' (a near-homophone error). Words that are considered misspelled given a certain context are called *context-sensitive spelling errors* or *context-dependent spelling errors*.

As a complement to traditional grammar checkers, several approaches have been proposed for the detection and correction of context-sensitive spelling errors. The algorithms define sets of easily confused words, called *confusion sets*. For example, 'their' is often confused with 'there' or 'they're'. To begin with, we locate all words in all confusion sets in our text. Given a word, the task for the algorithm is to determine which of the words in a confusion set is the most suitable in that position. To determine the most suitable word, several techniques have been used, such as Bayesian classifiers (Gale and Church, 1993; Golding, 1995; Golding and Schabes, 1996), Winnow (Golding and Roth, 1996), decision lists (Yarowsky, 1994), latent semantic analysis (Jones and Martin, 1997) and others. Golding and Roth (1999) report that the most successful method is Winnow with about 96% accuracy on determining the correct word for each confusion set.

In theory, when the spell checker, the grammar checker and the confusion set disambiguator have processed the text, only the unpredictable context-sensitive spelling errors remain. These are difficult to detect since they originate from random keyboard misspells producing real words. To approach this problem, we propose a transformation-based algorithm in Chapter 6, called ProbCheck. There, the text is compared to a corpus representing the "language norm". If the text deviates too much from the norm, it is probably ungrammatical, otherwise it is probably correct. If the method finds text that does not correspond to the norm, we try to transform rare grammatical constructions to those more frequent. If the transformed sentence is now close to the language norm, the original sentence was probably grammatically correct. The algorithm was evaluated in Chapter 11 and achieved acceptable results for this very difficult problem.

1.2 Evaluation

The performance of any NLP system (a grammar checker in the example above) depends heavily on the components it uses. For example, if the tagger has 95% accuracy, 5% of the words will receive the wrong PoS tag. If each sentence contains 10 words on the average, every second sentence will contain a tagging error. The tagging errors will in turn affect the parser. Also, the parser introduces errors of its own. If the parser has 90% accuracy, every sentence will contain one error on the average. This, in turn, will affect the grammar checker.

We see that the performance of the components of an NLP system affects the over-all performance in a complex way. Small changes in e.g. the tagging procedure or the noun phrase recognition affect large portions of the system. When modifying the system, to determine which changes are for the better, we need to *evaluate* the components and/or the system. Since many changes of the system components may result in many evaluations, manual evaluation is just not cost-efficient. A better approach is to let a human produce an annotated resource once, on which the evaluation is carried out. Thus, the standard setup for an evaluation is a supervised evaluation where the output of the NLP system is compared to a corpus annotated with the correct answers.

Even though we require a human to produce the resource, it is not unusual to use the NLP system as an aid in the annotation process. First, we apply the NLP system to a text and then, a human subject will correct the output. From this, we obtain an annotated resource. Unfortunately, starting out with the output of the NLP system might give the annotated resource a slight bias towards the starting data. Albeit, this is the most cost-efficient procedure to produce an annotated resource.

Repeated evaluation on the same annotated resource is not without its problems. The more the system's output is adjusted to imitate the annotated resource, the better the accuracy. We may obtain a system that has learned the idiosyncrasies of the resource, but lacks generality. Thus, when faced with a new, unknown text,

we obtain a much lower accuracy than we expected. To mitigate this problem, we divide the annotated resource into, say, ten parts. Normally, nine of them are used for training and tuning while one is used for testing. By using the test part very seldom, we do not over-fit our system to the test data. If the method to be evaluated is based on supervised (or unsupervised) learning, we can repeat the evaluation process ten times: each time we let one of the ten parts be the test data while training on the other nine. The system accuracy is the average of the ten evaluations. This is called *ten-fold testing*.

Comparing the output of a PoS tagger to the corpus tags is straightforward. Since there is one tag per word, we have obtained a correct answer if the tagger output equals the corpus tag. On the other hand, comparing parser output is not as easy. Here, the parser output may be partially correct when e.g. a phrase begins at the correct word but ends at the wrong word. One way to approach this is to treat parsing as we treat tagging as specified by the CoNLL chunking task (Tjong Kim Sang and Buchholz, 2000). For example, using the IOB format proposed by Ramshaw and Marcus (1995), an example sentence provides the following output:

```
I    NP-begin
saw  VP-begin
a    NP-begin
big  AP-begin | NP-inside
dog  NP-inside
```

In the IOB format, a phrase is defined by its beginning (e.g. NP-begin) and the subsequent words that are part of the phrase (said to be inside the phrase, e.g. NP-inside). There is no need for ending a phrase since the beginning of another phrase ends the previous. Furthermore, we denote nested phrases by a pipe (|) in this example. Thus, 'a big dog' in the above sentence has a corresponding bracket representation '[NP a [AP big] dog]'. Now, we are given the output of a parser:

```
I    NP-begin
saw  NP-begin
a    NP-begin
big  NP-inside
dog  NP-inside
```

We see that the parser output is incorrect for both the words 'saw' and 'big'. Hence, when measuring the overall accuracy of the parser, we carry out the same evaluation as the tagger evaluation above. If the parser output is not fully correct, it is considered incorrect. Thus, note here that the word 'big' is incorrectly parsed even though the output is partially correct. Evaluating parser accuracy for individual phrases is more complicated and is discussed in Section 8.3. The IOB format is further explained in Section 5.5.

Another widespread metric for evaluating parser accuracy is the Parseval (or the Grammar Evaluation Interest Group, GEIG) metric (Black et al., 1991; Grishman et al., 1992), based on comparison of phrase brackets. It calculates the precision

and recall by comparing the location of phrase boundaries. If a phrase in the NLP system output has the same type, beginning and end as a phrase in the annotated resource, it is considered correct. If, on the other hand, there is an overlap between the output and the correct answer, it is partially correct. This type of occurrences is called *cross-brackets*. Thus, we define

$$\text{Labeled precision} = \frac{\text{number of correct constituents in proposed parse}}{\text{number of constituents in proposed parse}} \quad (1.1)$$

$$\text{Labeled recall} = \frac{\text{number of correct constituents in proposed parse}}{\text{number of constituents in treebank parse}} \quad (1.2)$$

$$\text{Cross-bracket} = \text{number of constituents overlapping a treebank constituent without being inside} \quad (1.3)$$

For example, we have a sentence in the annotated resource:

```
[NP the
  man]
[VP walked]
[NP his
  dog]
```

The parser output is

```
[NP the
  man]
[VP walked]
[NP his]
[NP dog]
```

and we see that the output for 'his dog' differs from the annotated resource while 'the man' and 'walked' are correctly parsed. Thus, the precision is $2/4 = 50\%$, the recall is $2/3 = 67\%$ and no cross brackets are found. Despite the widespread use of the Parseval metric, it has obtained some criticism (see e.g. Carroll et al., 1998), since it does not always seem to reflect the intuitive notion of how close an incorrect parse is to the correct answer.

The Parseval evaluation scheme is devised for phrase constituent evaluations. A related evaluation procedure for dependency parsers is given by Collins et al. (1999). Furthermore, some metrics and methods are applicable to any parse structure (Lin, 1995, 1998; Carroll et al., 1998; Srinivas et al., 1996). In Chapters 8 and 9, we apply the row-based CoNLL evaluation scheme (Tjong Kim Sang and Buchholz, 2000) to both dependency output and phrase constituency in the IOB format. In Chapter 9, we perform an unsupervised comparative evaluation on different formalisms on the same text.

Supervised evaluation requires an annotated resource in the target language. Large corpora annotated with PoS tag data exist in most languages and thus, PoS taggers using supervised training are readily available. On the other hand,

annotated resources for parser evaluation, often denoted *treebanks*, are not as widely developed. For example, no large treebank exists for Swedish. Furthermore, even if there exists a treebank, its information may not be compatible with the output of the parser to be evaluated. Also, mapping parse information from one format to another is difficult (Hogenhout and Matsumoto, 1996).

Nevertheless, where annotated resources do exist, supervised methods may be applied. A supervised evaluation procedure for parser robustness is discussed in Chapter 8. In Chapter 11, we propose a semi-supervised evaluation procedure for the detection algorithm for context-sensitive spelling errors. As mentioned previously, the ProbCheck algorithm achieves acceptable results despite a very difficult problem.

Small, annotated resources of high quality can actually help the construction of a large resource by using a method called *bootstrapping* (see e.g. Abney, 2002). We start out with a small amount of information and use supervised learning to train a parser. This parser is now used to parse a larger amount of text. A human then checks the output manually. Again, the parser is trained supervised, now on the larger resource. Finally, the full-sized text is parsed using the parser and is checked by a human. The idea is that the accuracy and generality of the parser improves with each iteration and that the requirement for human interaction is kept to a minimum. This is called weakly supervised learning.

An alternative, less labor-intense approach to create a treebank is to train on the small resource, parse a larger text and then, without checking it manually, use the larger text to train the parser again. The idea is that a larger text will enable the parser to generalize so that idiosyncrasies from the small resource will be less prominent. Clearly, this alternative method is more error-prone than the weakly unsupervised. The word bootstrapping actually stems from the fact that we lift ourselves in our bootstraps.

From the discussion above, we see that even when using bootstrapping, the construction of an annotated resource of good quality requires manual labor. To avoid manual labor, if an annotated resource is not available for the target language, we have to resort to unsupervised methods (for an overview, see Clark, 2001). As discussed earlier, unsupervised methods operate on raw, unlabeled text, which makes them cheap and easily portable to other languages and domains. In Chapter 9, we propose an unsupervised evaluation procedure for parser robustness. An evaluation of the unsupervised evaluation procedure showed that the results were very accurate, with few exceptions.

To facilitate the design of unsupervised and supervised evaluation procedures, we have developed two generic tools called Missplel and AutoEval, described in Chapters 3 and 4, respectively. Their use is discussed in Section 7.2, as well as in the evaluation in Chapters 8 through 11. In the evaluation chapters, we found the tools very useful and time-saving in the development of unsupervised and other automatic evaluations.

1.3 Papers

This thesis is based upon work presented in the following papers:

I. (Bigert and Knutsson, 2002) Johnny Bigert and Ola Knutsson, 2002. Robust Error Detection: A hybrid approach combining unsupervised error detection and linguistic knowledge. In *Proceedings of Romand 2002*. Frascati, Italy.

II. (Bigert et al., 2003a) Johnny Bigert, Linus Ericson and Antoine Solis, 2003. AutoEval and Missplel: Two generic tools for automatic evaluation. In *Proceedings of Nodalida 2003*. Reykjavik, Iceland.

III. (Knutsson et al., 2003) Ola Knutsson, Johnny Bigert, and Viggo Kann, 2003. A robust shallow parser for Swedish. In *Proceedings of Nodalida 2003*. Reykjavik, Iceland.

IV. (Bigert et al., 2003b) Johnny Bigert, Ola Knutsson and Jonas Sjöbergh, 2003. Automatic evaluation of robustness and degradation in tagging and parsing. In *Proceedings of RANLP 2003*. Bovorets, Bulgaria.

V. (Bigert, 2004) Johnny Bigert, 2004. Probabilistic detection of context-sensitive spelling errors. In *Proceedings of LREC 2004*. Lisboa, Portugal.

VI. (Bigert et al., 2005b) Johnny Bigert, Jonas Sjöbergh, Ola Knutsson and Magnus Sahlgren, 2005. Unsupervised evaluation of parser robustness. In *Proceedings of CICLing 2005*. Mexico City, Mexico.

VII. (Bigert et al., 2005a) Johnny Bigert, Viggo Kann, Ola Knutsson, Jonas Sjöbergh, 2005. Grammar checking for Swedish second language learners. In *CALL for the Nordic languages 2005*. Samfundslitteratur.

VIII. (Bigert, 2005) Johnny Bigert, 2005. Unsupervised evaluation of Swedish spell checker correction suggestions. Forthcoming.

Papers **I** and **V** discuss the implementation of a detection algorithm for context-sensitive spelling errors. The algorithm is described in Chapter 6 and the evaluation of the algorithm is given in Chapter 11.

Paper **II** describes two generic tools for NLP system evaluation. They are explained in Chapters 3 and 4. Their use in supervised and unsupervised evaluation is described in Section 7.2, and they are used for evaluation purposes in Chapters 8 through 11.

Paper **III** elaborates on the implementation of a shallow parser for Swedish. It is discussed in Chapter 5 and is evaluated in Chapters 8 and 9.

Papers **IV** and **VI** address supervised and unsupervised evaluation of parser robustness. These topics are covered in Chapters 8 and 9.

Paper **VII** summarizes the work conducted in the CrossCheck project and includes some of the work mentioned above. It describes the use of the ProbCheck algorithm (from Chapter 6) in second language learning.

Paper **VIII** describes an unsupervised evaluation procedure for correction suggestions from spell checkers. The evaluation procedure and the results for Swedish are given in Chapter 10.

The author was the main contributor for articles **I**, **II**, **IV**, **V**, **VI** and **VIII**. That is, for these papers, the author developed the main idea and much of the software. For article **III**, the author wrote the parser software by interfacing the Granska framework and constructed the phrase selection heuristics. In paper **VII**, the author contributed with the ProbCheck algorithm.

1.4 Definitions and Terminology

For readers not fully accustomed to the terminology of NLP, we devote this section to defining the key concepts used in the rest of the thesis.

General Terminology

Natural language – Language produced by a human (e.g. written text or spoken language).

Natural language processing (NLP) – Computerized processing of natural language to deduce or extract information. For example, the spell checker in a word processing program.

NLP system – a program or a more complex combination of programs processing natural language.

Natural language resource (or resource for short) – Natural language in computer readable format (e.g. written text in a text file or spoken language in an audio file).

Annotated resource (or corpus) – Natural language resource with additional information (annotations), normally manually created or corrected to ensure correctness. An example of an annotated resource is a text with part-of-speech and morphological information for each word.

Techniques

Part-of-speech (PoS) category – A categorization that determines the use of a word in a sentence. For example, the part-of-speech category for a word may be noun, verb, pronoun etc. Also, while the part-of-speech category of the word 'boy' is noun, the part-of-speech category of the word 'saw' might be either noun or verb, depending on the context in which it is used.

PoS tag – Extra information assigned to each word about its part-of-speech (e.g. noun, verb, pronoun etc.) and morphological information (e.g. singular for a noun, present tense for a verb, etc.).

PoS tagging (or just tagging) – The task of assigning a PoS tag to each word in a text.

Parsing – The task of assigning a relation between the words of a sentence. For example, a phrase constituent parser identifies e.g. noun and verb phrases while a dependency parser assigns functional dependencies to words, such as main word, attribute, subject and object.

Shallow parsing vs. full parsing – Full parsing generates a detailed analysis of a sentence and constructs a parse tree. That is, all nodes (words) are subordinated another node, and a special node, denoted *the root*, is the top node. On the other hand, shallow parsers do not build a parse tree with a top node. Thus, some words may be left without analysis.

Evaluation

Manual evaluation – The evaluation procedure (or parts of it) is carried out by hand.

Automatic evaluation – The evaluation procedure does not require any manual work. However, it may operate on an annotated resource.

Supervised evaluation – An automatic evaluation procedure applied to a resource annotated with the correct answers.

Unsupervised evaluation – An automatic evaluation procedure applied to raw, unlabeled text.

Semi-supervised evaluation – Supervised evaluation implies that an annotated resource is used to determine if the output of an NLP system is correct. Thus, the annotated resource is normally annotated with the correct answers. In several chapters of this thesis, we make use of an annotated resource not annotated with the correct answers. Hence, these methods are not supervised in the common sense. We have chosen to denote them *semi-supervised*. The 'supervised' part of the word stems from the fact that it uses an annotated resource, created by a human. The 'semi' part stems from the fact that the annotated resource is not annotated with the correct answers and thus, we obtain information beyond the annotated resource.

Learning and Training

Unsupervised learning/training – Extracting information or patterns from raw, unlabeled text.

Supervised learning/training – Extracting information or patterns from a resource annotated with the data to be learned.

Semi-supervised learning/training – Extracting information or patterns from a resource *not* annotated with the data to be learned. For further details, see the definition of semi-supervised evaluation.

Weakly supervised learning/training – A procedure for iteratively increasing the accuracy: Start out with a small, annotated resource for supervised training of an NLP system. Then, apply the trained NLP system on a large, unlabeled text. Apply the supervised training algorithm on the larger annotated data and iterate. For better accuracy, manually check the output in each iteration. Weakly supervised training is often called bootstrapping.

Part I

Tools and Applications

Chapter 2

Introduction to Tools and Applications

This part of the thesis describes two tools (Chapters 3 and 4) and two applications (Chapters 5 and 6). This Chapter will cover some background and describe a few of the applications developed at the Department of Numerical Analysis and Computer Science at the Royal Institute of Technology.

2.1 Background

Manual evaluation of NLP systems is time-consuming and tedious. When assessing the overall performance of an NLP system, we are also concerned with the performance of the individual components. Many components will imply many evaluations. Furthermore, during the development cycle of a system, the evaluations may have to be repeated a large number of times. Sometimes, a small modification of a single component may be detrimental to overall system performance. Facing the possibility of numerous evaluations per component, we realize that manual evaluation will be very demanding.

Automatic evaluation is often a good complement to manual evaluation. Naturally, post-processing of manual evaluations, such as counting the number of correct answers, is suitable for automation. Implementation of such repetitive and monotonous tasks is carried out in the evaluation of almost all NLP systems. To support the implementation of these evaluations, we have constructed a program for automatic evaluation called AutoEval. This software handles all parts frequently carried out in evaluation, such as input and output file handling and data storage, and further simplifies the data processing by providing a simple but powerful script language. AutoEval is described in Chapter 3.

Automatic evaluation is not limited to the gathering and processing of data. We have developed another program, called Missplel, which introduces human-like errors into correct text. By applying Missplel to raw text, the performance

of an NLP system can be automatically assessed under the strain of ill-formed input. An NLP system's ability to cope with noisy input is one way of measuring its robustness. MISSPLEL is described in described in Chapter 4.

AUTOEVAL and MISSPLEL have been successfully used for unsupervised and supervised evaluation, as described in chapters 8 through 11. Both programs are freeware and the source code is available from the web site (Bigert, 2003).

In the subsequent chapters, we describe two applications developed at the Department of Numerical Analysis and Computer Science. In Chapter 5, we describe how a shallow parser and clause identifier was implemented in the GRANSKA framework (Domeij et al., 2000; Carlberger et al., 2005). GRANSKA is a natural language processing system based on a powerful rule language. The shallow parser has been used in several applications. In Chapter 6, we describe an algorithm for the detection of context-sensitive spelling errors called PROBCHECK. In PROBCHECK, the shallow parser was used to identify and transform phrases. The probabilistic error detection algorithm was developed as a complement to the grammar checker developed in the GRANSKA NLP framework. There, grammatical errors are detected using rules, while PROBCHECK is primarily based on statistical information retrieved by semi-supervised training from a corpus.

2.2 Applications and Projects at Nada

At the department of Numerical Analysis and Computer Science (Nada), several NLP systems have been developed. Here, we give a brief overview of the systems related to this thesis.

Granska – a grammar checker and NLP framework. GRANSKA is based on a powerful rule language having context-sensitive matching of words, tags and phrases and text editing possibilities such as morphological analysis and inflection. Examples of the GRANSKA rule language can be found in Section 5.3. GRANSKA includes its own HMM PoS tagger (Carlberger and Kann, 1999). GRANSKA has been used for the development of a grammar checker (Domeij et al., 2000) and a shallow parser (Knutsson et al., 2003).

Stava – a spell checker. STAVA (Domeij et al., 1994; Kann et al., 2001) is a spell checker with fast searching by efficient storage of the dictionaries in so-called *Bloom filters*. STAVA includes morphological analysis and processing of compound words, frequent in e.g. Swedish and German. It is evaluated in Chapter 10.

GTA – a shallow parser. GRANSKA Text Analyzer (GTA) (Knutsson et al., 2003) is a shallow parser for Swedish developed using the GRANSKA NLP framework. It also identifies clauses and phrase heads, both used in the detection of context-sensitive spelling errors in Chapter 6. The implementation of GTA is discussed in Chapter 5.

ProbCheck – a detection algorithm for context-sensitive spelling errors. ProbCheck (Bigert, 2004) is a probabilistic algorithm for detection of difficult spelling errors. It is based on PoS tag and phrase transformations and uses GTA for phrase and clause identification. It is discussed in Chapter 6.

Grim – a text analysis system. Grim (Knutsson et al., 2004) is a word processing system with text analysis capabilities. It uses Granska, Stava, GTA and ProbCheck and presents the information visually.

CrossCheck – language tools for second language learners. CrossCheck (Bigert et al., 2005a) is a project devoted to the development of language tools for second language learners of Swedish.

Chapter 3

AutoEval

As mentioned in the introduction, evaluation is an integral part of NLP system development. Normally, the system consists of several components, where the performance of each component directly influences the performance of the overall system. Thus, the performance of the components needs to be evaluated. All evaluation procedures have several parts in common: data input and storage, data processing and finally, data output. To simplify the evaluation of NLP systems, we have constructed a highly generic evaluation program, named AutoEval. The strengths of AutoEval are exactly the points given above: simple input reading in various formats, automatic data storage, powerful processing of data using an extendible script language, as well as easy output of data.

AutoEval was developed by Johnny Bigert and Antoine Solis as a Master's thesis (Solis, 2003). It was later improved by Johnny Bigert and Linus Ericson.

3.1 Related Work

Several projects have been devoted to NLP system evaluation, such as the EAGLES project (King et al., 1995), the ELSE project (Rajman et al., 1999) and the DiET project (Netter et al., 1998). Most of the evaluation projects deal mainly with evaluation methodology, even though evaluation software has often been developed to apply the methodology. For example, a PoS tag test bed was developed in the ELSE project. Also, the TEMAA framework (Maegaard et al., 1997) has produced a test bed denoted PTB. There, AutoEval could be used to perform the actual testing by automatically collecting the relevant data, such as the ASCC (automatic spell checker checker) described in (Paggio and Underwood, 1998). The existence and diversity of existing test beds are compelling arguments for the need of a general evaluation tool. Using AutoEval, creating a test bed is limited to writing a simple script describing the evaluation task. Thus, a general tool as AutoEval would have greatly simplified the implementation of such test beds.

Despite the large amount of existing evaluation software, we have not been able to find any previous reports on truly generic and easy-to-use software for evaluation. The large amount of evaluation software further supports the need for a generic tool like AutoEval.

3.2 Features

AutoEval is a tool for automatic evaluation, written in C++. The main benefits of this generic evaluation system are the automatic handling of input and output and the script language that allows us to easily express complex evaluation tasks.

When evaluating an NLP system using AutoEval, the evaluation task is described in an XML configuration file. The configuration file defines what input files to be used and what format they are given in. Currently, AutoEval supports plain-text and XML files. The system handles any number of input files.

The evaluation to be carried out is defined by a simple script language. Figure 3.1 provides an example. The objective of the example script is to read two files: the first is from an annotated resource with PoS tags and the second is the same text with artificially misspelled words inserted into 10% of the words. The latter was tagged using a PoS tagger. The PoS tags are to be compared to see how often the PoS tagger is correct and how often a PoS category (such as adjective) is confused with another (e.g. adverb).

Lines 1–4 are just initialization of the xml. Line 6 specifies a library of functions called `tmpl.xml`. It contains functions commonly used, for example, the `wordclass` function (used in lines 24–25). Lines 8–12 are the preprocessing step of the configuration. It will only be processed once. Lines 9–11 specify the files to be used. We open the file `suc.orig` with the original tags of the annotated resource. In the rest of the configuration file it will be denoted by an alias `annot`. Correspondingly, the file with the misspelled words and PoS tagger output `suc.missplel.10.tnt` will be denoted `tagged`. Furthermore, an output file named `suc.result.xml` will be produced. It is in xml format and is called `outfile` in the rest of the configuration file.

Lines 13–30 are the processing step. The commands given in the processing step are carried out for each row of the input files. First, we parse the input files using the `field` command at lines 14 and 15. In this case, we specify that we have two data fields separated by tabs (`"\t"`) and that a line ends with newline (`"\n"`). The data found is saved in variables called `word1` and `tag1` for the input file containing the annotations (`annot`). The data found in the misspelled file (`tagged`) is saved in variables called `word2` and `tag2`.

In line 16, we increase (`++`) a variable (`stat$total`) counting the total number of rows in the input files. The name of the variable is `total` and it resides in a group called `stat`. The use of groups simplifies the output, as explained later. Every thousand row, we output (`print`) the number of lines processed to report on the progress (lines 18–19).

In lines 20–21, we compare the two words read (`word1` and `word2`), and if they differ, we update a variable `stat$misspelled` counting the number of misspelled words. At line 22, we check if the tags read (`tag1` and `tag2`) differ. If so, we first extract the word-class from the PoS tags at lines 24 and 25. Then, a counter called `(:wcl1)$(:wcl2)` is updated. The name of the variable is `(:wcl2)`, which is in fact the contents of the variable `wcl2`. Thus, if `wcl2` contains `nn` as in noun, the name of the variable to be updated is `nn`. The same applies to the group called `(:wcl1)`. If the variable `wcl1` is e.g. `vb` as in verb, the group would be `vb`. Hence, in this example, a variable called `vb$nn` would be increased. Thus, line 26 actually counts how many times one word-class is mistagged and confused with another word-class. Line 27 counts the total amount of incorrect tags by updating the counter `stat$mistagged`, and line 28 counts the number of times a particular tag has been tagged incorrectly. For example, if the variable `tag1` contains the noun tag `nn.utr.sin.def.nom`, the counter variable named `nn.utr.sin.def.nom` will be increased by one.

The post-processing step in lines 31–33 outputs all groups, thus outputting all variables that have been created in the processing section. The configuration file in Figure 3.1 was applied to the annotated file in Figure 3.2 and the misspelled file in Figure 3.3. The resulting output is given in Figure 3.4.

The script language permits overloading of function names. That is, the same function name with different number of parameters will result in different function calls. If the basic set of functions is not sufficient, the user can easily add any C++ function to the system. Thus, there is no limit to the expressiveness of the script language. Furthermore, common tasks (e.g. calculating precision and recall or extracting the word class as seen in lines 24–25 in the example) that you use often can be collected in repository files where they can be accessed from all configuration files.

AutoEval processes about 100 000 function calls (e.g. `field`) per second, or about 2000 rows (words) of input per second for the example script given here.

```
 1  <?xml version="1.0" encoding="ISO-8859-1"?>
 2  <root xmlns="evalcfgfile"
 3      xmlns:xsi="http://www.w3.org/2001/XMLSchema"
 4      xsi:schemaLocation="evalcfgfile cfg.xsd">
 5      <templates>
 6          <libfile>tmpl.xml</libfile>
 7      </templates>

 8      <preprocess>
 9          infile_plain("annot", "suc.orig");
10          infile_plain("tagged", "suc.missplel.10.tnt");
11          outfile_xml("outfile", "suc.result.xml");
12      </preprocess>

13      <process>
14          field(in("annot"), "\t", "\n", :word1, :tag1);
15          field(in("tagged"), "\t", "\n", :word2, :tag2);

16          ++stat$total;
17          // progress report
18          if(stat$total % 1000 == 0)
19              print(int2str(stat$total) . " words");

20          if(:word1 != :word2)
21              ++stat$misspelled;
22          if(:tag1 != :tag2)
22          {
24              :wcl1 = wordclass(:tag1);
25              :wcl2 = wordclass(:tag2);
26              ++(:wcl1)$(:wcl2);
27              ++stat$mistagged;
28              ++tags$(:tag1);
29          }
30      </process>

31      <postprocess>
32          output_all_int(out("outfile"));
33      </postprocess>
34  </root>
```

Figure 3.1: AUTOEVAL *configuration example counting the number of tags and the number of word-classes confused for another word-class.*

```
Men       kn                                (But)
stora     jj.pos.utr/neu.plu.ind/def.nom    (large)
företag   nn.neu.plu.ind.nom                (companies)
som       kn                                (such as)
Volvo     pm.nom                            (Volvo)
och       kn                                (and)
SKF       pm.nom                            (SKF)
har       vb.prs.akt.aux                    (has)
ännu      ab                                (not)
inte      ab                                (yet)
träffat   vb.sup.akt                        (struck)
avtal     nn.neu.plu.ind.nom                (deals)
.         mad                               (.)
```

Figure 3.2: *Example from an annotated file from the SUC corpus.*

```
Men       kn                                (But)
stora     jj.pos.utr/neu.plu.ind/def.nom    (large)
företag   nn.neu.plu.ind.nom                (companies)
som       hp                                (such as*)
Volvo     pm.nom                            (Volvo)
och       kn                                (and)
SKF       pm.nom                            (SKF)
har       vb.prs.akt.aux                    (has)
ännu      ab                                (not)
inge      vb.inf.akt                        (Inge/induce*)
träfat    nn.neu.plu.ind.nom                (wooden plate*)
avtal     nn.neu.sin.ind.nom                (deals*)
.         mad                               (.)
```

Figure 3.3: *Example of PoS tagger output on a file with misspelled words. Asterisks mark a tag or spelling discrepancy from Figure 3.2.*

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<evaloutput date="Wed Jul 16 16:16:54 2004">
    <ab>
        <var name="ab">7</var>
        <var name="dt">4</var>
        <var name="ha">5</var>
        ...
    </ab>

    <dt>
        <var name="ab">8</var>
        <var name="dt">10</var>
        <var name="jj">6</var>
        ...
    </dt>

    ...

    <stat>
        <var name="misspelled">1528</var>
        <var name="mistagged">2165</var>
        <var name="total">14119</var>
    </stat>

    <tags>
        <var name="ab">133</var>
        <var name="ab.kom">8</var>
        <var name="ab.pos">30</var>
        <var name="ab.suv">7</var>
        <var name="dt.mas.sin.ind/def">1</var>
        <var name="dt.neu.sin.def">20</var>
        ...
        <var name="nn.utr.sin.def.nom">26</var>
        ...
    </tags>
</evaloutput>
```

Figure 3.4: *Example output from* AUTOEVAL *when applying the configuration file in Figure 3.1 to the files in Figures 3.2 and 3.3.*

Chapter 4

Missplel

During the development of spell and grammar checkers such as Stava and Granska (briefly described in Section 2.2), we require a test text for evaluation. Preferably, the text should contain errors for the NLP system to detect. Unfortunately, resources annotated with information on spelling and grammatical errors are rare and time-consuming to produce. Furthermore, it may be difficult to detect all errors in a text and classify the errors found. Also, the data may be exhaustively used giving the system a bias towards the evaluation text. Nevertheless, these resources are often useful or required when evaluating spelling checkers and grammar checking systems as well as other NLP system performance under the influence of erroneous or noisy input data.

Presumably, conventional corpus data is well proof read and scrutinized and thus, it is assumed not to contain errors. To produce an annotated text with spelling and grammatical errors, we created a piece of software called Missplel. Missplel introduces artificial, yet human-like, spelling and grammatical errors into raw or annotated text. This will provide us with the exact location and type of all errors in the file.

This chapter reports on the features and implementation of Missplel. Examples of how the software is used are found in Section 7.2. There, we also determine the prerequisites for unsupervised versus supervised use of the tools.

Missplel was developed by Johnny Bigert and Linus Ericson as a Master's thesis (Ericson, 2004).

4.1 Related Work

Several sources report on software used to introduce errors to existing text. Most of these deal mainly with performance errors or so-called Damerau-type errors, i.e. insertion, deletion or substitution of a letter or transposition of two letters (Damerau, 1964).

For example, Grudin (1981) has conducted a study of Damerau-type errors made by typists and from that, implemented an error generator. Agirre et al. (1998) briefly describe AntiSpell that simulates spelling errors of Damerau type to evaluate spell checker correction suggestions. The results of Agirre et al. (1998) are further discussed in Chapter 10. Peterson (1986) introduced Damerau type spelling errors in a large English dictionary to establish how many words are one Damerau-type error away from another. He found that for a 370 000 word dictionary, 216 000 words could be misspelled for another word. The resulting words corresponded to 0.5% of all misspellings possible by insertion, deletion, substitution and transposition. Most of the misspelled words were a result of a substituted letter (62%).

Another error introducing software, ErrGen, has been implemented in the TEMAA framework (Maegaard et al., 1997). ErrGen uses regular expressions at letter level to introduce errors, which allows the user to introduce Damerau-type errors as well as many competence errors, such as sound-alike errors (*receive, recieve*) and erroneously doubled consonants. ErrGen was used for automatic spelling checker evaluation (Paggio and Underwood, 1998) and is further discussed in Chapter 10.

The features of all these systems are covered by Missplel. Furthermore, it offers several other features as well as maximum configurability.

4.2 Features

The main objective in the development of Missplel was language and PoS tag set independency as well as maximum flexibility and configurability. To ensure language and PoS tag independence, the language is defined by a dictionary file containing word, PoS tag and lemma information. The character set and keyboard layout are defined by a separate file containing a *distance matrix*, that is, a matrix holding the probability that one key is pressed instead of another.

Missplel introduces most types of spelling errors produced by human writers. It introduces performance errors and competence errors at both letter and word level by using four main modules: Damerau, SplitCompound, SoundError and SyntaxError. The modules can be enabled or disabled independently. For each module, we can specify an error probability. For example, if the Damerau module is set to a 10% probability of introducing an error, about 10% of the words in the text will be misspelled with one Damerau-type spelling error.

The Missplel configuration file, provided in XML, offers fine-grained control of the errors to be introduced. Most values in the configuration file will assume a default value if not provided. The format of all input and output files, including the dictionary file, is configurable by the user via settings using regular expressions.

Normally, misspelling 'cat' to 'car' would not be detected by a spelling or grammar checker. In Missplel, you can choose not to allow a word to be misspelled into an existing word or, if you allow existing words, choose only words that have

```
Letters NN2
would   VM0
be      VBI
welcome AJ0-NN1

Litters NN2  damerau/wordexist-notagchange
would   VM0  ok
bee     NN1  sound/wordexist-tagchange
welcmoe ERR  damerau/nowordexist-tagchange
```

Figure 4.1: MISSPLEL *example. The first part is the input consisting of row-based word/tag pairs. The second part is the* MISSPLEL*ed output, where the third column describes the introduced error.*

a different PoS tag in the dictionary. This information (whether the error resulted in an existing word and if the tag changed or not) can be included in the output as shown in the example in Figure 4.1.

The Damerau Module introduces performance errors due to keyboard mistypes (e.g. *welcmoe*), often referred to as Damerau-type errors. The individual probabilities of insertion, deletion, substitution and transposition can be defined in the configuration and are equally probable by default. In the case of insertion and substitution, we need a probability of confusing one letter for another. This distance matrix is provided in a separate file and simply contains large values for keys close to each other on the keyboard.

The Split Compound Module introduces erroneously split compounds. These errors are common in compounded languages like Swedish or German and may alter the semantics of the sentence. As an example in Swedish, 'kycklinglever' ('chicken liver') differs in meaning from 'kyckling lever' ('chicken is alive'). A multitude of settings are available to control the properties (e.g. length and tag) of the first and second element of the split compound.

The Sound Error Module introduces errors the same way as ERRGEN mentioned in Section 4.1, that is, by using regular expressions at letter level. In MISSPLEL, each rule has an individual probability of being invoked. This allows common spelling mistakes to be introduced more often. Using the regular expressions, many competence errors can easily be introduced (e.g. misspelling 'their' for 'there').

The Syntax Error Module introduces errors using regular expressions at both letter and word/tag level. For example, the user can form new words by modifying the tag of a word. The lemma and PoS tag information in the

dictionary help Missplel to alter the inflection of a word. This allows easy introduction of feature agreement errors ('he are') and verb tense errors such as 'sluta skrik' ('stop shout'). You can also change the word order, double words or remove words.

The foremost problems with resources annotated with errors are, for most languages, availability and the size of the resources. Using Missplel, the only requirement is a resource annotated with word and PoS tag information, available for most languages. From this, we can create an unlimited number of texts with annotated and categorized errors.

Missplel uses randomization when introducing errors into a text to be used for evaluation of the performance of an NLP system. To reduce the influence of chance on the outcome of the evaluation, we may run the software repeatedly (say, n times) to obtain any number of erroneous texts from the same original text. The average performance on all texts will provide us with a reliable estimate on the real performance. The standard deviation should also be considered. Low standard deviation would imply that the average is a good estimate on the real performance. Note here that the number of iterations n does not depend on the size of the annotated resource.

Missplel processes about 1000 rows (words) of input per second for the parser robustness evaluation in Chapter 8.

Chapter 5

GTA – A Shallow Parser for Swedish

In many NLP-applications, the robustness of the internal modules of an application is a prerequisite for the success and usefulness of the system. The full spectrum of robustness is defined by Menzel (1995), and further explored according to parsing by Basili and Zanzotto (2002). In our work, the term robustness refers to the ability to retain reasonable performance despite noisy, ill-formed and partial natural language data. For an overview of the parser robustness literature, see e.g. Carroll and Briscoe (1996).

In this chapter, we will focus on a parser developed for robustness against ill-formed and partial data, called Granska Text Analyzer (GTA).

5.1 Related Work

When parsing natural language, we first need to establish the amount of details required in the analysis. Full parsing is a very detailed analysis where each node in the input receives an analysis. Evidently, a more detailed analysis opens up for more errors. If we do not require a full analysis, shallow parsing may be an alternative. The main idea is to parse only parts of the sentence and not build a connected tree structure and thus limiting the complexity of the analysis.

Shallow parsing has become a strong alternative to full parsing due to its robustness and quality (Li and Roth, 2001). Shallow parsing can be seen as a parsing approach in general, but also as pre-processing for full parsing. The partial analysis is well suitable for modular processing which is important in a system that should be robust (Basili and Zanzotto, 2002). A major initiative in shallow parsing came from Abney (1991), arguing both for psycholinguistic evidence for shallow parsing and also its usability in applications for real world text or speech. Abney used hand-crafted cascaded rules implemented with finite state transducers. Current research in shallow parsing is mainly focused on machine learning techniques (Hammerton et al., 2002).

An initial step in shallow parsing is dividing the sentence into base level phrases, called text chunking. The Swedish sentence 'Den mycket gamla mannen gillade mat' ('The very old man liked food') would be chunked as:

```
(NP Den mycket gamla mannen)(VP gillade)(NP mat)
(NP The very old man)(VP liked)(NP food)
```

The next step after chunking is often called phrase bracketing. Phrase bracketing means analyzing the internal structure of the base level phrases (chunks). NP bracketing has been a popular field of research (e.g. Tjong Kim Sang, 2000). A shallow parser would incorporate more information than just the top-most phrases. As an example, the same sentence as above could be bracketed with the internal structure of the phrases:

```
(NP Den (AP mycket gamla) mannen)(VP gillade)(NP mat)
(NP The (AP very old) man)(VP liked)(NP food)
```

Parsers for Swedish

Early initiatives on parsing Swedish focused on the usage of heuristics (Brodda, 1983) and surface information as in the Morp Parser (Källgren, 1991). The Morp parser was also designed for parsing using very limited lexical knowledge.

A more complete syntactic analysis is accomplished by the Uppsala Chart Parser (UCP) (Sågvall Hein, 1982). UCP has been used in several applications, for instance in machine translation (Sågvall Hein et al., 2002).

Several other parsers have been developed recently. One uses machine learning (Megyesi, 2002b) while another is based on finite-state cascades, called Cass-Swe (Kokkinakis and Johansson-Kokkinakis, 1999). Another parser (Nivre, 2003) assigns dependency links between words from a manually constructed set of rules. A parser based on the same technique as the previous is called Malt (Nivre et al., 2004) and uses a memory-based classifier to construct the rules. Both Cass-Swe and Malt also assigns functional information to constituents.

There is also a full parser developed in the Core Language Engine (CLE) framework (Gambäck, 1997). The deep nature of this parser limits its coverage.

Furthermore, two other parsers identify dependency structure using Constraint Grammar (Birn, 1998) and Functional Dependency Grammar (Voutilainen, 2001). These two parsers have been commercialized. The Functional Dependency parser actually builds a connected tree structure, where every word points at a dominating word.

Several of these parsers are used and further discussed in Chapter 8.

5.2 A Robust Shallow Parser for Swedish

GTA is a rule-based parser for Swedish and relies on hand-crafted rules written in the Granska rule language (Carlberger et al., 2005). The rules in the grammar

are applied on PoS tagged text, either from an integrated tagger (Carlberger and Kann, 1999) or from an external source. GTA identifies constituents and assigns phrase labels. However, it does not build a full tree with a top node.

The basic phrase types identified are adverbial phrases (ADVP), adjective phrases (AP), infinitival verb phrases (INFP), noun phrases (NP), prepositional phrases (PP), verb phrases (VP) and verb chains (VC). The internal structure of the phrases is parsed when appropriate and the heads of the phrases are identified. PP-attachment is left out of the analysis since the parser does not include a mechanism for resolving PP-attachments.

For the detection of clause boundaries, we have implemented Ejerhed's algorithm for Swedish (Ejerhed, 1999). This algorithm is based on context-sensitive rules operating on PoS tags. One main issue is to disambiguate conjunctions that can coordinate words in phrases, whole phrases and, most important, clauses. About 20 rules were implemented for the detection of clause boundaries in the Granska framework.

The parser was designed for robustness against ill-formed and fragmentary sentences. For example, feature agreement between determiner, adjective and noun is not considered in noun phrases and predicative constructions (Swedish has a constraint on agreement in these constructions). By avoiding the constraint for agreement, the parser will not fail due to textual errors or tagging errors. Tagging errors that do not concern agreement are to some extent handled using a set of tag correction rules based on heuristics on common tagging errors.

5.3 Implementation

To exemplify the rules in Granska, we provide an example of a feature agreement rule from the Granska grammar scrutinizer in Figure 5.1. First, X, Y and Z are words. For a word to be assigned to X, it has to fulfill the conditions given in brackets after X. In this case, the word class has to be a determiner (`dt`). The same applies to Y, where the word has to be an adjective (`jj`). Furthermore, Y can contain zero or more consecutive adjective, denoted with a star (`*`). Last, Z has to be a noun (`nn`) and it has to have a feature mismatch with X: either the gender, the number (`num`) or the species (`spec`) mismatch.

If such a sequence of words is found, the left-hand side of the rule has been satisfied. The arrow (`-->`) separates the left-hand side of the rule from the right-hand side. The left hand-side of the rule is the conditions to be fulfilled. The right-hand part of rule is the action to take when the conditions have been fulfilled.

In this case, we mark the words found (`mark`) for the user, suggest a correction (`corr`) by modifying the features on X (the determiner) to agree with Z (the noun). A hint is also supplied to the user (`info`).

As seen from the example in Figure 5.1, the rules consist of several PoS tags or PoS tag categories to be matched. In the example, we specify that the first word is a determiner (`dt`), which is in fact a collection of 13 tags such as `dt.utr.sin.def`

```
disagree@incongruence
{
  X(wordcl=dt),
  Y(wordcl=jj)*,
  Z(wordcl=nn &
      (gender!=X.gender |
       num!=X.num | spec!=X.spec))
-->
  mark(X Y Z)
  corr(X.form(gender:=Z.gender,
              num:=Z.num, spec:=Z.spec))
  info("The determiner" X.text
       "does not agree with the noun" Z.text)
  action(scrutinizing)
}
```

Figure 5.1: *An example of the* Granska *rule language*

and `dt.neu.plu.ind` (see Table 7.1 for an explanation of the tag set). Clearly, if none of these tags are present in a sentence, applying the rule is a waste of time. On the other hand, if the tags are very frequent, the rule will be applied too frequently.

A better approach to determine when a rule is applicable is to use bigrams of PoS tags. In the rule example, we know that a determiner must be followed by either an adjective (`jj`) or, if there are no adjectives, a noun (`nn`). Thus, possible sequences of tag categories are '`dt nn`', '`dt jj nn`', '`dt jj jj nn`' etc. We see that the possible pairs of tag categories are '`dt nn`', '`dt jj`', '`jj jj`' and '`jj nn`'. Thus, for the example rule to apply, one of these bigrams must occur in the text. Granska automatically determines all possible PoS tag pairs for all rules and stores them in a table. After a sentence has been tagged, the PoS tag pairs are looked up in the table and the appropriate rules are applied.

As an example of a GTA rule, we provide a verb chain help rule in Figure 5.2. A help rule can be applied from other rules. The example would match a sentence such as 'har han inte gått' ('has he not left'). The first word X is 'ha/har/hade' ('has/have/had'). It has to be followed by an noun phrase (`NP`) which in turn has to be followed by an optional adverbial chain (`ABCHAIN`). The `NP` and `ABCHAIN` are themselves help rules. After that, the verb (`vb`) Y is matched having supine form (`sup`). The line beginning with `action` sets the return value from the help rule to verb in preterite form (`prt`) in the same voice as the verb Y.

```
VBCHAIN_NP_VB@
{
  X(text="ha" | text="har" | text="hade"),
  (NP)(),
  (ABCHAIN)()?,
  Y(wordcl=vb & vbf=sup),
-->
  action(help, wordcl:=vb, vbf:=prt, voice:=Y.voice)
}
```

Figure 5.2: *Example of a GTA help rule.*

5.4 The Tetris Algorithm

The phrase recognition rules of GTA are very much similar to the example given in the previous section. When applying the rules, the Granska rules output all possible phrases found. Parts of a sentence may not have an analysis and some parts of a sentence will have received overlapping phrases. Thus, we require a means to disambiguate the phrases found.

We have developed a heuristics for disambiguating the phrases obtained from Granska called The Tetris algorithm. As the reader may know, Tetris is a game where different sized (and shaped) blocks fall from above. The aim of the game is to fit the blocks into a space in the bottom of the screen.

In the Tetris algorithm, the phrases are the blocks to be fitted into a space. We start out with the largest phrases (those spanning the most words). First, we adopt the right-most phrase as a part of the parser output. We then proceed to the left placing the phrases with the same length. When all phrases of a certain length are used, we proceed to the next shorter length and start over filling from right to left. The criterion to fulfill is that no phrase must cross the beginning or end of another phrase. They may be inside another or adjacent but we must never have a phrase having only one of two end points inside another phrase.

As an example of the Tetris algorithm, consider the sentence 'Jag (I) pratar (am talking) med (to) Peter (Peter) Olsson (Olsson)'. The phrases found by the GTA parser are

- 'med Peter Olsson' (to Peter Olsson), PP, length 3
- 'Peter Olsson' (Peter Olsson), NP, length 2
- 'med Peter' (to Peter), PP, length 2
- 'pratar' (am talking), VP, length 1

- 'Jag' (I), NP, length 1

Here, the items of the list are sorted in the order they will be attempted, with the longest and right-most phrases first. The first list item is a prepositional phrase spanning the last three words of the sentence. It is the first phrase and thus, it cannot cross any boundaries, so it is accepted. Since there are no more phrases of length 3, we carry on with phrases of length 2. The first out is the right-most phrase 'Peter Olsson'. We make sure that it does not cross the boundaries of the phrases placed so far. Since it shares one boundary with the first phrase and has its other boundary inside the first phrase, it is also accepted. However, the third phrase 'med Peter' (to Peter) overlaps the second phrase since its rightmost boundary is inside the second phrase while the leftmost boundary is outside. Thus, the third phrase is discarded. The length one phrases cannot cross a boundary and are all accepted, resulting in

```
Jag    (I)          NPB
pratar (am talking) VPB
med    (to)         PPB
Peter  (Peter)      NPB|PPI
Olsson (Olsson)     NPI|PPI
```

The output format is explained in the next section. Bracketed, the result is '[NP Jag][VP pratar][PP med [NP Peter Olsson]]'.

5.5 Parser Output

```
Viktigaste   APB|NPB  CLB  (the most important)
redskapen    NPI      CLI  (tools)
vid          PPB      CLI  (in)
ympning      NPB|PPI  CLI  (grafting)
är           VCB      CLI  (is)
annars       ADVPB    CLI  (normally)
papper       NPB|NPB  CLI  (paper)
och          NPI      CLI  (and)
penna        NPB|NPI  CLI  (pen)
,            0        CLB
menade       VCB      CLI  (meant)
han          NPB      CLI  (he)
.            0        CLI
```

Figure 5.3: *Example sentence showing the IOB format.*

```
((CL (NP (AP Viktigaste) redskapen)
    (PP vid (NP ympning))
    (VC är)
    (ADVP annars)
    (NP (NP papper) och (NP penna)))
(CL  ,
    (VC menade)
    (NP han)) . )
```

Figure 5.4: *The text from Figure 5.3 in a corresponding bracketing format.*

The output from the GTA parser is provided in the so-called IOB format (Ramshaw and Marcus, 1995). See Figure 5.3 and 5.4 for a sentence with phrase labels and clause boundaries in the IOB and bracketing format, respectively. As an example, `NPB|PPI` means that the beginning (`B`) of a noun phrase (`NP`) is within (`|`) the inside (`I`) of a prepositional phrase (`PP`). Thus, the rightmost phrase is the topmost node in the corresponding parse tree. `CLB` and `CLI` are the beginning and inside of a clause, respectively. The phrase types were explained in Section 5.2.

Chapter 6

ProbCheck – Probabilistic Detection of Context-Sensitive Spelling Errors

Algorithms for the detection of misspelled words have been known since the early days of computer science. The program would simply look up a word in a dictionary, and if not present there, it was probably misspelled. Unfortunately, not all misspelled words result in an unknown word. Misspelled words resulting in existing words are called context-sensitive spelling errors, since a context is required to detect an error. Clearly, these errors are much more problematic than normal spelling errors since they require at least a basic analysis of the surrounding text.

In this chapter, we propose a transformation-based probabilistic algorithm for the detection of context-sensitive spelling errors. The algorithm is based on supervised learning and PoS tag and phrase transformations.

6.1 Related Work

Several approaches have been proposed to address context-sensitive spelling errors. To detect commonly confused words (e.g. *there, they're, their*), methods using confusion sets have been proposed as discussed in the introduction in Section 1.1 (e.g. Yarowsky, 1994; Golding, 1995; Golding and Roth, 1996). They use a limited set of errors, either manually constructed or obtained automatically. These algorithms are useful for the detection of frequently occurring spelling errors. Unfortunately, context-sensitive spelling errors due to words outside the confusion set will not be processed. The algorithm proposed here is able to process and detect any misspelled word.

Another approach used transition probabilities and error likelihoods from PoS taggers (Atwell, 1987). Unfortunately, these probabilities were not very reliable as seen in the evaluation in Chapter 11. Also, an approach using supervised learning of errors to train a classifier was described in (Sjöbergh and Knutsson, 2004). However, machine learning is more suitable for predictable errors such as split compounds and

verb tense errors. The relation between the classifier and the proposed algorithm is discussed in Chapter 11.

Full parsing would be the ideal solution to detect context-sensitive spelling errors. The words that do not fit into the grammar are misplaced. To achieve reasonable accuracy for a full parser, an extensive amount of manual work is required. Furthermore, the processing of the text will be difficult if there are several errors in the same region since the parser will have little or no context to base its analysis upon. The method proposed here requires much less manual work and is very robust to multiple errors. In Chapter 11, we compare the parser approach to the ProbCheck algorithm.

6.2 PoS Tag Transformations

In this section, we present the probabilistic method for detection of spelling errors, not requiring any previous knowledge of error types. The algorithm is based upon statistics from a corpus of correct text.

Automation and Unsupervision

The definition of semi-supervised learning in Section 1.4 implies that an algorithm is trained on an annotated resource not explicitly containing the data to be learned. In this chapter, we obtain information used for error detection from a PoS tagged corpus with no errors. Thus, the training is not supervised in the normal sense, and we denote it semi-supervised.

Detection of Improbable Grammatical Constructs

Part-of-speech tag n-grams have many useful properties. As the n-grams are extracted from a corpus representing the language, they capture some of the language's features. Because of the limited scope of an n-gram, the extracted features will contain only local information. Each of these n-grams constitutes a small acceptance grammar since it describes an acceptable sequence of n PoS tags in the language. Altogether, the n-grams form a grammar containing local information about the acceptable grammatical constructs of the language. In contrast, PoS tag n-grams not in the grammar may be an indication of ungrammaticality. From these observations, we will construct a first, naive error detection algorithm. An implementation for trigrams is shown in Algorithm 1.

The text to be scrutinized must first be tagged with a PoS tagger. From the resulting tag stream, each n-gram is looked up in a table holding the frequency of each n-gram obtained from a corpus. If the frequency exceeds a pre-determined threshold, the construct is considered grammatically sound. Otherwise, it is a rare or incorrect grammatical construct, and therefore improbable to be the intention of the writer. Thus, the n-gram is flagged as a potential grammatical error.

Algorithm 1: NaiveProbCheck
Description: A first approach to a probabilistic error detector
Input: A tag stream $\bar{s}_k = (t_1, t_2, \ldots, t_k)$ and a grammaticality threshold e.
Output: A set of indexes of ungrammatical constructs if found, $\emptyset$ (the empty set) otherwise.

```
ProbCheck(s̄_k, e)
(1)   I ← ∅
(2)   foreach i in [2, k − 1]
(3)     if TrigramFreq(t_{i−1}, t_i, t_{i+1}) < e
(4)       I ← I ∪ {i}
(5)   return I
```

One serious problem concerning this approach is rare constructs due to insufficient data and infrequent tags. An n-gram representing an acceptable grammatical construct may not have been encountered because of the rareness of the tags participating in the n-gram.

Sparse Data and PoS Tag Transformations

We note that rare PoS tags often result in rare tag n-grams and use an example to illustrate the problem with rare grammatical constructs.

Say that we have encountered a sentence in Swedish 'Det är varje chefs uppgift att...' (It is every manager's responsibility to...). The tag disambiguator has tagged the part 'det är varje' (it is every) with (`pn.neu.sin.def.sub/obj, vb.prs.akt, dt.utr/neu.sin.ind`). (See Table 7.1 for an explanation of the tag set.) A consultation of the trigram frequency table reveals that this particular trigram has never been encountered before even though the construction is grammatically sound. This may be attributed to the fact that one of the participating tags has low frequency and in this example, the third tag (`dt.utr/neu.sin.ind`) is rare with only 704 occurrences (0.07% out of a million words). A language construct, very much similar in meaning to the one above, is 'det är en' (it is a) with tags (`pn.neu.sin.def.sub/obj, vb.prs.akt, dt.utr.sin.ind`). This small change in meaning increases the individual tag frequency from 704 occurrences for (`dt.utr/neu.sin.ind`) to 19112 occurrences for (`dt.utr.sin.ind`). The trigram frequency rises from 0 occurrences for (`pn.neu.sin.def.sub/obj, vb.prs.akt, dt.utr/neu.sin.ind`) to 231 occurrences for (`pn.neu.sin.def.sub/obj, vb.prs.akt, dt.utr.sin.ind`). We see that replacing (`dt.utr/neu.sin.ind`) with (`dt.utr.sin.ind`) reduces the problem with rare tags while retaining almost the same meaning. The sentence becomes 'Det är en chefs uppgift att...' (It is a manager's responsibility to...).

The example indicates that we could benefit from substituting a rare tag with a tag of higher frequency suitable in the same context. Thus, we transform the PoS

tags of the sentence to obtain a more frequent sequence.

Clearly, when transforming a PoS tag into another, not all tags are equally suitable. We require a distance between two tags, or put differently, a probability for one tag being suitable in the place of another. One approach to produce such a distance is to use a norm.

A norm is a measure of the size of an entity. In our case, we apply the norm to the difference between two probability distributions, one from each PoS tag. There are several different norms, and one of them is the L1 norm, $L1(P_1, P_2) = \sum_{d \in D} |P_1(d) - P_2(d)|$. Thus, the L1 norm is the absolute difference between all points in the definition set D. In our case, D is the set of all PoS tag trigrams as we will see below. We use trigrams as an example of how the distance is calculated for PoS tag n-grams.

We are given a PoS tag trigram (t_L, t, t_R). If we want to transform t into another tag t', we first need to know how suitable t' is in the place of t. We denote t_L the left context and t_R the right context. After the transformation we will have a new PoS trigram (t_L, t', t_R).

Our first observation is that PoS tags of high frequency yield high frequency trigrams on the average. Hence, to be able to compare the frequencies of a trigram containing t and a trigram containing t' we need to compensate for their difference in frequency. To this end, we normalize the frequency of the trigram (t_L, t, t_R):

$$\widetilde{freq}(t_L, t, t_R) = \frac{freq(t_L, t, t_R)}{freq(t)}.$$

We note that $\widetilde{freq}(t_L, t, t_R) \in [0, 1]$. Second, we calculate the difference between the normalized frequencies of the two tags:

$$dist_{t_L, t_R}(t, t') = \left| \widetilde{freq}(t_L, t, t_R) - \widetilde{freq}(t_L, t', t_R) \right|.$$

Now, we have a distance between two tags t and t' given a fixed context t_L and t_R. We want to determine how suitable t' is in the place of t given any context. Thus, we need to consider all PoS tag contexts:

$$dist(t, t') = \sum_{t_L, t_R} dist_{t_L, t_R}(t, t').$$

Here, we make a few observations. Since the distance measure is based upon the L1 norm, it is a metric. This means that first, $dist(t, t') \geq 0$. Next, $dist(t, t') = 0$ if and only if $t = t'$. Translated into PoS tag terminology this means that if the uses of two tags are identical, the two tags are the same, since there is no difference between the trigram frequencies. Last, $dist(x, y) + dist(y, z) \geq dist(x, z)$, which is the triangle inequality saying that the distance from one tag to another via a third is a longer distance than going from the first directly to the second.

Furthermore, we establish an upper bound for the distance:

$$\begin{aligned} dist(t,t') &= \sum_{t_L,t_R} \left| \widetilde{freq}(t_L,t,t_R) - \widetilde{freq}(t_L,t',t_R) \right| \leq \\ &\leq \sum_{t_L,t_R} \left| \widetilde{freq}(t_L,t,t_R) \right| + \left| \widetilde{freq}(t_L,t',t_R) \right| \leq 2 \end{aligned}$$

Thus, $dist(t,t')$ ranges from 0 (where the contexts are identical) to 2 (where the uses of t and t' are disjoint).

As a further refinement of the data extraction from the corpus, we consider using the distance not only from the second position of the trigram, but also the first and third. The astute reader may notice that this may result in an overlap between two trigrams. That is, if a PoS tag is found in the corpus, there are three trigrams overlapping it. Using the frequency of all three trigrams will result in a slight over-estimation of the true frequency. The size of the tag set $|\mathrm{T}|$ will determine how much a reused context tag will influence the estimation. The error will be proportional to $1/|\mathrm{T}|$ which is small with a reasonable sized tag set. Nevertheless, only the middle of the trigram was used here during the distance calculation to simplify the exposition.

To simplify the description of the algorithm in the next section, we choose to express the distances as values between zero and one. We will denote them "probabilities" to simplify the exposition. Thus, we define the probability of a successful transformation to be $p(t,t') = 1 - dist(t,t')/2$, that is, the probability is 0 for totally disjoint syntactic uses and 1 for the same tag. We call p the transformation success probability.

Weighted n-grams

Given the distances, we now have the tools to transform rare tags to those more common. When a tag trigram of low frequency is encountered, we want to determine whether the low frequency is due to ungrammaticality or merely the low frequency of the participating tags. Hence, we want to determine whether substituting one of the tags may increase the frequency. When transforming a tag into another, we must take into consideration the syntactic distance between the tags involved, when calculating the new trigram frequency resulting from the switch.

For example, given the trigram (t_1,t_2,t_3) with frequency $f = freq(t_1,t_2,t_3)$, we use the tag t_1' to replace t_1. Note that the distances are calculated using only the middle tag, while a transformation may occur in any tag.

From the distance discussion we get a probability of $q = p(t_1,t_1')$. A q of 1 would imply that t_1 and t_1' are used in identical syntactic contexts and thus, no penalty should be imposed. A $q < 1$ implies that the use of t_1 and t_1' differs, and a penalty is in order since there is a probability that the use of t_1' in this context may be less appropriate than the use of t_1. We calculate the new trigram frequency for (t_1,t_2,t_3) as $f' = freq(t_1',t_2,t_3) \cdot q$, that is, the new trigram frequency

penalized. If f' is above a given frequency threshold, thus improving on the old trigram frequency, the construct is considered grammatically sound.

When substituting more than one tag simultaneously we take into consideration all syntactic distances involved by defining the compound penalty $p(t_1, t'_1) \cdot p(t_2, t'_2) \cdot p(t_3, t'_3)$. Keep in mind here, that when replacing a tag with itself, the success probability is one and thus no penalty is incurred.

We now construct a measure of the probability of grammaticality when given a trigram tag sequence. The intention here is to consider several possible transformations for each of the three tags in the trigram. (Note that the tag itself is included among the attempted transformations.) We choose to limit the number of possible replacements for each PoS tag to the m tags having the highest probability.

Thus, we have m different tags in three positions resulting in m^3 new trigrams. Given the new trigrams, we calculate a compound trigram frequency involving the new trigrams and their penalties.

Definition: *The* weighted trigram frequency *of a trigram sequence* (t_1, t_2, t_3) *is defined as*

$$wfreq(t_1, t_2, t_3) = \sum_{t'_1, t'_2, t'_3} p(t_1, t'_1) \cdot p(t_2, t'_2) \cdot p(t_3, t'_3) \cdot freq(t'_1, t'_2, t'_3),$$

where the sum is over all the m^3 *different combinations of substitute tags.*

The intuition behind the weighted frequency is simply to attempt all different combinations of replacements for the tags in the trigram. We will use the weighted frequency as a measurement of the grammaticality of a sentence. For each of the trigrams in the sentence we apply the weighted frequency function *wfreq* and if it is below a given threshold, that part is considered ungrammatical. Note that the original trigram is among the new trigrams.

Clearly, we could use other means to combine the penalized frequencies. For example, the maximum of the terms in the *wfreq* sum was evaluated as well as several other functions. Nevertheless, summation obtained the best performance.

The algorithm

The final algorithm is implemented for trigrams in Algorithms 2 and 3. Algorithm 2 is very similar to Algorithm 1 but utilizes weighted trigrams. In Algorithm 3, the compound penalty is computed over the m^3 representatives. For each representative, the penalties are computed on lines 8–10 and the trigram frequency at line 11. From these, the weighted frequency is calculated at line 12.

6.3 Phrase Transformations

The main problem with the probabilistic error detection is the fact that phrase and clause boundaries may produce almost any PoS tag n-gram and thus, many

Algorithm 2: ProbCheck
Description: The improved probabilistic error detector
Input: A tag stream $\bar{s} = (t_1, t_2, \ldots, t_k)$ and a grammaticality threshold e
Output: A set of indexes where the ungrammatical constructs are found, $\emptyset$ (the empty set) otherwise

```
ProbCheck(s̄, e)
(1)    I ← ∅
(2)    foreach i in [2, k − 1]
(3)        if WeightedTrigramFreq(t_{i−1}, t_i, t_{i+1}) < e
(4)            I ← I ∪ {i}
(5)    return I
```

Algorithm 3: WeightedTrigramFreq
Description: Calculate weighted trigram frequencies
Input: A tag trigram (t_1, t_2, t_3)
Output: The weighted trigram frequency of the trigram provided

```
WeightedTrigramFreq(t1, t2, t3)
(1)    sum ← 0
(2)    T'1 ←ClosestTags(t1)
(3)    T'2 ←ClosestTags(t2)
(4)    T'3 ←ClosestTags(t3)
(5)    foreach t'1 in T'1
(6)        foreach t'2 in T'2
(7)            foreach t'3 in T'3
(8)                p1 ← p(t1, t'1)
(9)                p2 ← p(t2, t'2)
(10)               p3 ← p(t3, t'3)
(11)               f ← freq(t'1, t'2, t'3)
(12)               f' ← p1 p2 p3 f
(13)               sum ← sum + f'
(14)   return sum
```

n-grams have never been encountered. In this section, we make use of phrases and clause boundaries to remove false alarms resulting from such boundaries.

Clause and Phrase Recognition

The identification of clause and phrase boundaries is important for syntactic analysis. For example, the recognition of clause boundaries is an essential and repeated step in Constraint Grammar parsing (Karlsson et al., 1995). We have chosen to implement a rule-based phrase and clause identifier (see Chapter 5), even though

a parser using supervised learning would suffice. However, the parser used here is also capable of identifying phrase heads, which makes the supervised learning more demanding. The most important quality of the parser is robustness.

We want to transform long and rare phrases to the more common, minimal phrases consisting of the head only. The replacement of a phrase results in a longer scope for the PoS tagger and thus, a longer scope for the probabilistic error detector.

The module for phrase recognition identifies the phrase candidates and assigns them with the head's feature values. For example, a noun phrase of the type 'den lilla pojken som sitter där borta' (the little boy sitting over there) is assigned with the following features and values: word class is noun, gender is non-neuter, number is singular, species is definite, case is nominative. This results in a valid tag (`nn.utr.sin.def.nom`), corresponding to the head 'pojken' (the boy).

The transformation must result in one or more valid tags to be useful to the probabilistic error detection algorithm. Furthermore, some constructs may be removed (replaced with zero tags) from the analyzed text (e.g. prepositional and adverbial phrases), which is motivated by the observation that removal of such phrases seldom violates the syntax of the language. For example, in the sentence 'I saw him in London' the prepositional phrase could be removed giving us 'I saw him'. In the sentence 'You have a very nice car', the adverbial phrase is removed giving us 'You have a nice car'. Although the meaning of the sentence is slightly changed, the syntax is not violated.

The rules implemented are liberal regarding the syntactic agreement within the phrase. We have chosen this strategy for several reasons. First, we want to analyze sentences that may contain one or more errors. Second, the linguistic rules for agreement in Swedish contain some problematic exceptions. Third, tagging errors from the part-of-speech tagger could cause insufficient disambiguation of phrase boundaries.

Applying Phrase Transformations

Our aim is to produce a sentence without rare n-grams while retaining grammaticality and preferably meaning similar to the original sentence. As explained in the previous section, every phrase may be replaced by another phrase having one or more tags (e.g. noun and verb phrases). A phrase may also be removed resulting in zero tags (e.g. adverbial or prepositional phrases).

An implementation of the phrase enhanced probabilistic error detection for trigrams is given in Algorithm 4. At line 3, the tag stream is probabilistically checked for grammatical errors. If no errors are found in any part of the sentence, the sentence is considered grammatical. The clause boundary condition is checked at line 4 so that detections adjacent to a clause boundary are not reported as errors. If no clause boundary is found, we turn to phrases and phrase boundaries. In line 5, the phrases are transformed to establish if the error was due to a rare phrase construction.

Algorithm 4: PhraseProbCheck
Description: The phrase enhanced probabilistic error detector
Input: A tag stream $\bar{s} = (t_1, t_2, \ldots, t_k)$ and a grammaticality threshold e.
Output: A set of indexes of ungrammatical constructs if found, $\emptyset$ (the empty set) otherwise.

```
PhraseProbCheck(s̄, e)
(1)   I ← ∅
(2)   foreach i in [2, k − 1]
(3)     if WeightedTrigramFreq(t_{i−1}, t_i, t_{i+1}) < e
(4)       if not ClauseBoundary(i, s̄)
(5)         if not TransformOk(i, s̄, e)
(6)           I ← I ∪ {i}
(7)   return I
```

In Algorithm 5, we seek to resolve the problem with the rare trigrams found due to phrase boundaries. At line 1 we identify the phrases overlapping the trigram at index i. From these, we construct all combinations of phrases such that no two phrases span a common PoS tag index (line 2). In each of the combinations, we replace the participating phrases with their heads (line 4), or if it is a prepositional or adverbial phrase, we remove the phrase. From each combination of transformations, we have obtained a new sentence. If the trigram at index i in the new tag stream $\bar{s}' = (t'_1, t'_2, \ldots, t'_k)$ is approved by the probabilistic error detection (line 5), we consider the trigram grammatically sound. If none of the combinations result in an acceptable PoS tag trigram, a grammatical error is reported at line 7.

Algorithm 5: TransformOk
Description: The algorithm for phrase replacement and removal
Input: An index i containing a rare trigram, a tag stream $\bar{s} = (t_1, t_2, \ldots, t_k)$ and a grammaticality threshold e
Output: True if the trigram is grammatical, False otherwise

```
TransformOk(i, s̄, e)
(1)   P ← FindOverlappingPhrases(i, s̄)
(2)   C ← PhraseCombinations(P)
(3)   foreach C in C
(4)     s̄' ← ReplaceWithHeads(C, s̄)
(5)     if WeightedTrigramFreq(t'_{i−1}, t'_i, t'_{i+1}) ≥ e
(6)       return True
(7)   return False
```

The use of the algorithm is best illustrated with an example. Say that we have encountered the sentence ‘den (the) lilla (little) vasen (vase) på (on) hyllan (the

shelf) är (is) inte (not) så (so) ful (ugly)' where the part 'hyllan är inte' (shelf is not) is tagged (`nn.utr.sin.def.nom, vb.prs.akt.kop, ab`). The initial probabilistic test erroneously indicates an error. (See Table 7.1 for an explanation of the tag set.)

We construct the phrases overlapping the trigram centered at index 6 (see Figure 6.1):

A NP: den lilla vasen på hyllan (the little vase on the shelf) →
vasen (`nn.utr.sin.def.nom`) (the vase)

B PP: på hyllan (on the shelf) → *remove*

C ADVP: inte så (not so) → *remove*

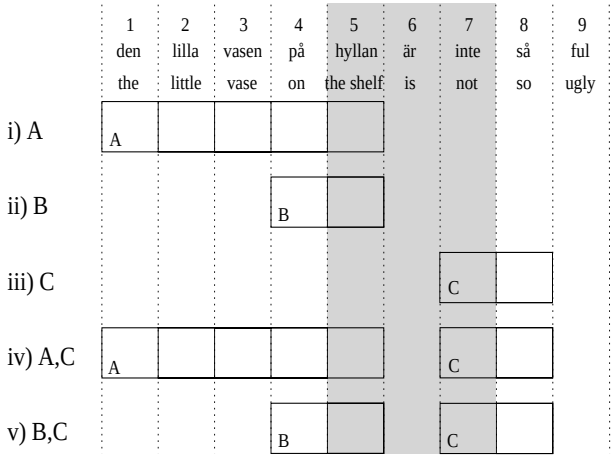

Figure 6.1: *Combination of phrases overlapping the suspicious trigram (highlighted).*

The single word 'är' ('is') cannot be transformed and is ignored. From the phrases A, B and C, we construct all combinations as shown in Figure 6.1. The combination (A, B) is not included due to the overlap between the two phrases. The resulting sentences are shown in Figure 6.2. Combinations **i** and **ii** both produce rare trigrams due to the adverbial construction 'inte så' (not so). Combination **iii** removes the adverbial construction and produces an acceptable trigram.

Throughout the replacements, the algorithm attempts to retain grammaticality, even though the content of the sentence may be somewhat altered, as seen in Figure 6.2. Note that there is a probability that any transformation, PoS or phrase, yields an ungrammatical construction. Hence, the algorithm is called ProbCheck.

i) vasen är inte så ful **(the vase is not so ugly)**	vasen (the vase)	6	7	8	9		
ii) den lilla vasen är inte så ful **(the little vase is not so ugly)**	1	2	3	6	7	8	9
iii) den lilla vasen på hyllan är ful **(the little vase on the shelf is ugly)**	1	2	3	4	5	6	9
iv) vasen är ful **(the vase is ugly)**	vasen (the vase)	6	9				
v) den lilla vasen är ful **(the little vase is ugly)**	1	2	3	6	9		

Figure 6.2: *The resulting sentences from the combinations in Figure 6.1.*

6.4 Future Work

The PoS tag transformation approach yielded high recall but low precision. To increase the precision, phrase transformations were used. However, this reduced the recall considerably. Another approach could be considered to increase the precision of the PoS tag transformations. Originally, the PoS tag distances were extracted from the corpus without any use of the context. We attempted an approach where the left PoS tag was used as context. Thus, given a PoS tag as left context, we got the probability of successfully substituting a tag for another. As an effect, we obtained 149 tables resembling the original table, one for each PoS tag left context. This lead to problems with sparse data, and the 149 tables were reduced to 14, representing the 14 different word-classes as left context. To represent the right context, 14 more tables were created.

Hence, when a difficult PoS tag sequence has been detected, we choose one of the tags t to be replaced. The tag to the left of t will serve as the left context. We extract the word-class of the left context tag and consult the corresponding word-class table to see which tags are the most suitable as a replacement of t, giving us a list of e.g. ten candidates. As an example, say that we have encountered the difficult trigram (`nn.utr.sin.def.nom, vb.prs.akt.kop, ab`). We want to replace the center tag. Using context, the trigram would give us `nn` as left context since the tag to the left is (`nn.utr.sin.def.nom`). Now, (`vb.prs.akt.kop`) is looked up in a table where the PoS tag distance data was collected from the corpus only where `nn` was found as left-context. Thus, the table is context-sensitive and should have a higher relevance. Alas, the data will also be sparser.

The right context could also be used, giving us another ten candidates. In preliminary tests conducted, the overlap between the left-context list and the right-context list was limited. Furthermore, the ordering of the candidates varied a lot because of the probabilities given from the left and right context. Thus, different

weighting schemes to incorporate the left and right context could be considered. We could choose to ignore the right context, but the differences between the information given from the left and right context indicated that using only the left context would be an over-simplification. Note also that the original PoS tag distances use both left and right context when extracting information about the center tag of a trigram. The use of context-sensitive substitution of PoS tags would probably improve the results from the proposed method and would certainly be an interesting topic for future work.

The ProbCheck algorithm does not categorize the errors found, nor does it present a correction suggestion. However, the categorization would probably benefit the user of a word processing system using ProbCheck. Writing rules for categorization of the errors found would probably fail due to the unpredictable nature of the detected errors. A different approach would be to use machine learning in an attempt to learn patters originating from different error categories. To obtain the required material, artificial errors could be introduced into error-free text. This would be suitable for error types such as split compounds and missing words. A similar approach to error detection has been proposed by Sjöbergh and Knutsson (2004). There, machine learning is used to train an error detector on artificial errors. However, in our case, the most problematic errors, such as context-sensitive spelling errors, would probably be out of reach for such an algorithm. Nevertheless, this would be an interesting issue for future work.

Part II

Evaluation

Chapter 7

Introduction to Evaluation

The second part of this thesis discusses four different evaluation methods. The first two concern parser robustness evaluation. The next compares the performance of Swedish spell checkers. The last evaluates the performance of the context-sensitive spelling error detection algorithm. Some of the evaluations made use of annotated resources, discussed in the next section. The evaluations were automated using two tools as explained in Section 7.2.

7.1 The Stockholm-Umeå Corpus

The experiments in Chapters 8 through 11 all require proof-read text. We have chosen to adopt the text from the Stockholm-Umeå corpus (SUC) (Ejerhed et al., 1992). The SUC is a balanced collection of written Swedish, annotated with PoS tag information and it contains about one million words. The part-of-speech tag set contains 149 tags, such as `dt.utr.sin.ind` or `vb.prs.akt` (see Table 7.1 for an explanation). Originally, SUC did not contain any parse information. We annotated a portion of the corpus with parse information in order to evaluate the GTA parser (from Chapter 5). We chose six texts (aa02, ac04, je01, jg03, kk03 and kk09) from three different text categories for a total of 14 000 words (about a thousand sentences). The categories were press articles (a), scientific journals (j) and fiction (k). The texts were first run through the GTA parser and then carefully corrected by a human annotator. The tokenization and sentence boundaries were determined by the corpus.

7.2 Using Missplel and AutoEval in Evaluation

As stated in Chapter 2, manual evaluation of NLP systems is tedious and time-consuming. The use of Missplel and AutoEval proposed in this section can greatly reduce the amount of manual work required, if not totally eliminate it.

noun (nn)	pronoun (pn)	verb (vb)	determiner (dt)	adverb (ab)
non-neuter (utr)	neuter (neu)	singular (sin)	plural (plu)	
definite (def)	indefinite (ind)	nominative (nom)	genitive (gen)	
present (prs)	active (akt)	copula (kop)		
subject (sub)	object (obj)			

Table 7.1: *Examples of the features from the tag set used. The tag set comprises 149 tags. Examples: 'springer' (runs) is* vb.prs.akt *and 'bilens' (the car's) is* nn.utr.sin.def.gen.

Motivation

Unrestricted text often contains spelling and grammatical errors as well as missing, transposed and doubled words. The ability to handle input having these properties is one important aspect of the robustness of an NLP system.

To evaluate the robustness of an NLP system, we want to simulate the kind of noisy and malformed input presented to the system during normal use. As explained in Chapter 4, resources annotated with spelling errors are rare. Thus, we have chosen another approach to obtain noisy NLP system input. We start out from an arbitrary text (preferably well-written and proof-read) and introduce artificial errors using Missplel.

Introduction of artificial noise has been proposed earlier in the context of neural networks and language engineering (Miikkulainen, 1996; Mayberry, 2004), where the weights of the neural network were disturbed to simulate noise. Introduction of background noise in phone conversations was carried out in the Aurora experimental framework (Pearce and Hirsch, 2000). There, samples of different amounts of background noise, such as 'crowd of people' or 'street', were added to the conversation. Agirre et al. (1998) and Paggio and Underwood (1998) introduce artificial spelling errors in order to evaluate spell checkers. Despite the existence of automatic spell checker evaluation, we have not found any references in the literature indicating use of noise introduced in text to facilitate the design of automatic evaluation of other fields, such as parser robustness.

Clearly, there are many types of malformed input such as regular spelling errors, context-sensitive spelling errors, grammatical errors (e.g. split compounds and feature disagreement), missing and repeated words, unfinished sentences, hesitations, restarts etc. Nevertheless, we have chosen to use spelling errors to simulate noisy input for several reasons. First, performance (keyboard) spelling errors are language independent. Hence, anyone can use the proposed evaluation procedures and apply them to their parser in their language without modification. Second, performance spelling errors are easily described and widely understood and thus, do not obscure the important parts of the evaluation procedure. Also, to keep the description of the error model as straightforward as possible, we have refrained from applying an

automatic spelling corrector. Furthermore, evaluation in Section 8.6 showed that automatic correction of spelling errors actually resulted in lowered performance for the NLP system!

Please keep in mind that the evaluation methods in the subsequent chapters are not restricted to spelling errors, but applicable to any error type. For example, they could be used to evaluate parser robustness facing incomplete sentences (missing words), in the sense of e.g. Vilares et al. (2003), Lang (1988) and Saito and Tomita (1988).

Introducing Keyboard Mistypes

As explained in Section 4.2, keyboard mistype spelling errors most often result in so-called Damerau-type errors: a deleted letter, an inserted letter, a letter replaced by another or two letters transposed (switched places). We chose to distribute these four types equally among the introduced errors.

When a user presses the wrong key, not all keys are equally probable. Keys closer to the intended key are clearly more probable to press by mistake than one further away. The probability of hitting one key instead of another was determined by the distance between the center of the two keys. To avoid tokenization synchronization problems, we avoided introducing spaces and other delimiters into words.

Now, to introduce spelling errors, we started out with an error-free text. MISSPLEL was configured to randomly insert errors in a given percentage of the words. In the following chapters, we introduced errors in 1%, 2%, 5%, 10% and 20% of the words. Only one error was introduced into a misspelled word.

The intended use of the resulting misspelled text determines the kind of spelling errors to be introduced. For example, when evaluating a standard spell checker, it would be unfair to introduce spelling errors resulting in existing words. In the parser evaluations (Chapters 8 and 9) and spell checker evaluation (Chapter 10), MISSPLEL was configured to introduce errors resulting in non-existing words only. In the ProbCheck evaluation (Chapter 11), MISSPLEL was configured to introduce errors resulting in existing words only, and furthermore, the resulting word was required to have a different PoS tag than the original word. Please refer to the individual chapters for more information on the evaluation procedure.

Reducing the Influence of Chance

Hopefully, random introduction of errors provides a fair distribution of errors in terms of difficulty, length etc. An additional benefit of introducing artificial errors to an error-free text is the fact that the original text can be reused over and over again.

To reduce the influence of chance, we chose to introduce errors ten times per error level (1%, 2%, 5%, 10% and 20%), thus giving us fifty different misspelled texts. The NLP system to be evaluated was applied to each of the misspelled texts and AUTOEVAL was used to perform the evaluation on each output. Hence, for

a given error level, we obtained ten different results. Again using AutoEval, we calculated the mean and standard deviation for the ten files at each error level. The result was five files, one per error level, containing the mean and standard deviation of the performance for the NLP system.

Automation and Unsupervision

We note that introducing errors resulting in non-existing words is a fully unsupervised procedure. The dictionary required to determine if a word is existing or not can be built unsupervised from large amounts of text. This fact was exploited in the parser evaluation in Chapters 8 and 9 and the spell checker evaluation in Chapter 10. On the other hand, the introduction of errors in the ProbCheck evaluation (Chapter 11) required a list of PoS tags for each word. Thus, this dictionary could not be obtained by unsupervised learning. Nevertheless, it was built automatically from a PoS tagged corpus.

Note also that the evaluation in Chapter 8 was not labeled unsupervised due to the use of an annotated resource in another part of the evaluation process. The introduction of errors was still unsupervised.

Chapter 8

Supervised Evaluation of Parser Robustness

In this chapter, we present an automatic evaluation method focusing on the accuracy and robustness of parsers for syntactic analysis. The robustness of a parser is defined here as robustness against ill-formed input such as spelling errors, which is only one of the aspects of robustness as pointed out by Menzel (1995). The proposed method uses Missplel to introduce different kinds of errors into a text. The errors can be any type of spelling or grammatical errors, but we have focused on keyboard mistype spelling errors for reasons explained in Section 7.2. Furthermore, we introduce only spelling errors resulting in non-existing words to avoid some ambiguity problems, as explained later.

To demonstrate the evaluation method, it was applied on a shallow parser for Swedish. The experiments are presented as a glass box evaluation, where the performance of the over-all system is presented as well as the performance of the components, such as part-of-speech taggers. All tests were conducted with various levels of errors introduced, under which the system performance degradation was measured.

Since this chapter focuses on evaluation methodology, we do not address how the introduced errors affect the syntactic structure. Nevertheless, automatic evaluation of the effects on syntactic structure is indeed an interesting topic for future work.

8.1 Automation and Unsupervision

In the proposed method, we use a treebank to evaluate parser robustness when faced with noisy input, such as spelling errors. However, the treebank does not contain any errors. Normally, supervised evaluation implies that an annotated resource is used. That is, we apply the parser on the treebank text and compare the output to the treebank parse information. In the evaluation procedure proposed here, we reuse the parse information but introduce noise in the treebank text to evaluate

parser robustness. Nevertheless, since we use the parse information as found in the treebank, the evaluation procedure is denoted supervised.

Parts of this chapter address parser accuracy. These parts are supervised in the normal sense. That is, we apply the parser on the original text of the treebank.

8.2 Related work

Automatic parsing of text is a popular field of research. Many of the applications where parsing is used, such as parsing human input to a computer system, handle text that is not proofread. Depending on the application, the text can be relatively error free (e.g. parsing newspaper articles from the internet) or contain large amounts of errors (e.g. using a parser as a tool for second language learners when writing essays). If the intended use of a parser is domains with many errors, it must be robust enough to produce useful output despite noisy input. It is not sufficient to achieve a good performance on error-free text.

Evaluating Parsers

Carroll et al. (1998) give a comprehensive overview of different parser evaluation methods and discuss some shortcomings. Evaluation of parsers is usually carried out by comparing the parser output to a manually annotated or manually corrected version of a parsed test text. Manual work is expensive, and not necessarily error free. If the NLP system is under development, the evaluation has to be carried out repeatedly. Thus, very large amounts of annotated resources may be required to avoid data exhaustion. Many languages have no large manually annotated resources at all, and those existing often contain only error-free texts.

Manual annotation is not only expensive, but also hard to reuse when evaluating a new parser with a different grammar. Generally, it is non-trivial to map the output of one parser to the output of another (Hogenhout and Matsumoto, 1996). Often, different parsers do not generate the same information, so a mapping would have to add or remove information. Thus, the effort of manually annotating text with one type of parse information is generally not reusable for other parsers.

Robustness Evaluation

Robustness in this context is defined as the system's reluctance to change its output when the input becomes increasingly noisy and ill-formed. There are, as pointed out by Menzel (1995), many other types of robustness. To name a few examples, Basili and Zanzotto (2002) have proposed an evaluation procedure for robustness when faced with increasing language complexity. Vilares et al. (2004) use a small subset of English and define robustness as the ability to produce a reasonable amount of parse trees when the possible number of parse trees grows fast.

Concerning robustness against noisy text such as spelling errors, Li and Roth (2001) use the Penn Treebank and the Switchboard corpus. The latter serves as a

treebank with noisy text while the former is supposedly error-free. The advantage of this approach is that the data used is authentic. The drawback is that authentic data is rare, since noisy treebank data is unavailable for many languages. Furthermore, the texts are from different genres, which makes the comparison difficult. In this case, the Penn Treebank is based on Wall Street Journal articles while the Switchboard is transcribed phone calls. The proposed method is applicable to any treebank text, which makes it applicable to any language having an (error-free) treebank. Furthermore, the evaluation of robustness is carried out on the same text as the evaluation of accuracy on error-free text, which makes the comparison of the results straight-forward.

Another approach is proposed by Foster (2004). There, a treebank containing noisy text is manually corrected to serve as an error-free text. This approach eliminates the problems with different texts having different characteristics. However, manual work is required and the problem with access to a noisy corpus remains.

The method proposed here does not require a resource containing noisy text. Such resources are rare and do not exist in many languages. The proposed method uses only a regular treebank containing error-free text and thus, it is applicable to most languages.

8.3 Proposed Method

We wanted to assess the robustness of parsers when applied to noisy and malformed input. As stated in Chapter 7.2, there are many types of noise, but to provide an example of the proposed evaluation method, we have chosen to focus on spelling errors.

Introducing Spelling Errors

As noted in Chapter 4, resources annotated with noisy and malformed language are rare. To overcome this problem, we followed the procedure in Chapter 7.2 and used Missplel to introduce artificial spelling errors to correct text.

Misspelling a word into another, already existing word, may have the effect of altering the original interpretation of the sentence. This is indeed a problem since the parse tree of the new sentence may differ from that of the original sentence. Thus, there is a possibility that the output of the parse system is in fact correct even though it differs from the annotated parse tree. We approach this problem by restricting the introduced errors to spelling errors that result in non-existing words only. Hence, the new sentence does not have a straightforward interpretation. Nevertheless, the most plausible interpretation of the new sentence is that of the original text.

Phrase Recall and Precision

As discussed in Section 5.5, the output of the GTA parser is given in the IOB format. Using AutoEval, we gathered information on tag accuracy, full row parse accuracy, clause boundary identification accuracy as well as precision, recall and F-scores for all phrase types.

In order to calculate recall and precision for different phrase types, we needed a way to extract information from the IOB format. To keep the evaluation model simple, we did not consider partially correct answers. Thus, the statistics for individual phrase types were calculated as follows. We have a parse output from an NLP system. Given a phrase type to evaluate, all other phrase types were removed from the parse output. The same was done for the correct, annotated parse, and the results were then compared. The parser was successful if and only if they were identical. For example, we are looking at phrases of type NP. If the correct parse is `APB|NPB|NPI` (an adjective phrase in a noun phrase inside another noun phrase), the parse `NPB|APB|NPI` would be correct since the adjective phrase is ignored in both parses, while the parse `APB|NPI|NPI` would be incorrect since the leftmost NP differs.

Baseline Comparison

Since many parsers rely heavily on the performance of a part-of-speech tagger, we included several taggers with different behavior and characteristics. Apart from taggers representing state-of-the-art in part-of-speech tagging, we also included a perfect tagger and a baseline tagger. The perfect tagger did nothing more than copy the original tags found in the annotated resource. The baseline tagger was constructed to incorporate a minimal amount of linguistic knowledge and was included to establish the difficulty of the tagging task.

Parsing different texts may result in different accuracy for the parser at hand. To provide a clue to the inherent difficulty of a text, we required a baseline for the parsing task. The perfect tagger, the baseline tagger and the baseline parser are further discussed in the experiments section.

In the experiments below, we used five error levels (1%, 2%, 5%, 10%, 20%) as well as the error-free text (0% errors). For a given error level p, we introduced spelling errors (resulting in non-existing words only) in a fraction p of the words. This procedure was repeated ten times to mitigate the influence of chance and to determine the standard deviation of the accuracy and F-scores. The F-score is defined as

$$F_\beta = \frac{(1+\beta^2)\cdot \text{precision}\cdot \text{recall}}{\beta^2\cdot \text{precision} + \text{recall}}, \tag{8.1}$$

where β determines how important precision is in relation to recall. Here, we use $\beta = 1$ meaning that precision and recall is equally important.

With increasing amounts of errors in the text, the performance of the parser will degrade. In order to be robust against ill-formed and noisy input, we want the

accuracy to degrade gracefully with the percentage of errors. That is, for a parser relying heavily on PoS tag information, we aim for the parsing accuracy to degrade equal to or less than the percentage of tagging errors introduced. Of course, this is not feasible for all phrase types. For example, when the infinite marker or verb is misspelled, an infinitival verb phrase will be difficult to identify.

8.4 Experiments

We used MISSPLEL and AUTOEVAL to evaluate the rule-based GTA parser for Swedish, as described in Chapter 7.2. For this purpose, we annotated a part of the Stockholm-Umeå corpus (SUC) with parse information as described in Section 7.1.

We compared tagged text from four different sources: the original corpus tags, a hidden Markov model (HMM) tagger, a transformation-based tagger and a baseline tagger. The tagger CORPUS used the original annotations in the SUC corpus, which we assume to have 100% accuracy. The HMM tagger used was TnT (Brants, 2000), hereafter denoted TNT. The transformation-based tagger (Brill, 1992) used was fnTBL (Ngai and Florian, 2001), denoted BRILL. The baseline tagger called BASE chose the most frequent tag for a given word and, for unknown words, the most frequent tag for open word classes. All taggers were trained on SUC data not included in the tests.

To determine the difficulty of the chosen texts, we constructed a baseline parser. To this end, we adopted the approach provided by the CoNLL chunking competition (Tjong Kim Sang and Buchholz, 2000), i.e. for a given part-of-speech tag, the parse chosen was the most frequent parse for that tag. Given the PoS tagged text, the data was divided into ten parts. Nine parts were used for training. The last part was used for evaluation. With ten different possible evaluations, the performance of the base-line parser was the average of the ten evaluations. Furthermore, to determine the difficulty of the clause boundary identification we devised a baseline clause identifier simply by assigning a clause beginning (`CLB`) to the first word of each sentence and `CLI` to the other words. The clause identification output is described in Chapter 5.

Thus, we had three taggers (BASE, BRILL, TNT) and two parsers (GTA and baseline). For each combination of tagger and parser, we ran ten tests at each error level (1%, 2%, 5%, 10% and 20%) and one test on the error-free text (0%). Also, the CORPUS tagger was used with the baseline and GTA parsers. In each test, we extracted information about tagging accuracy, parsing accuracy, clause boundary identification and phrase identification for the individual phrase categories ADVP, AP, INFP, NP, PP and VC. Also, since some tokens are outside all phrases, we included an outside category (O). The phrase types are explained in Chapter 5.

8.5 Results

An important aspect of the accuracy of the GTA parser is the performance of the underlying tagger. Most taggers were quite robust against ill-formed and noisy input as seen from Table 8.1. For example, at the 20% error level, TnT degraded 13.1% and Brill degraded 15.2% relatively to their initial accuracy of 95.8% and 94.5%, respectively. The low degradation in performance is most likely due to the robust handling of unknown words in Brill and TnT, where the suffix determines much of the morphological information. Thus, if the last letters of a word are unaffected by a spelling error, the tag is likely to remain unchanged. The robustness of the baseline tagger was not as satisfactory as it guessed the wrong tag in almost all cases (19.0% of 20%). The baseline tagging accuracy for text without errors was 85.2%.

Tagger	0%	1%	2%	5%	10%	20%
Base	85.2	84.4 (0.9)	83.5 (1.9)	81.2 (4.6)	77.1 (9.5)	69.0 (19.0)
Brill	94.5	93.8 (0.7)	93.0 (1.5)	90.9 (3.8)	87.4 (7.5)	80.1 (15.2)
TnT	95.8	94.9 (0.9)	94.3 (1.6)	92.4 (3.5)	89.4 (6.7)	83.2 (13.1)

Table 8.1: *Accuracy in percent from the tagging task. The* Corpus *tagger was assumed to have 100% accuracy. The columns denote the amount of errors introduced. Relative accuracy degradation compared to the 0% error level is given in brackets.*

Tagger	0%	1%	2%	5%	10%	20%
Base	81.0	80.2 (0.9)	79.1 (2.3)	76.5 (5.5)	72.4 (10.6)	64.5 (20.3)
Brill	86.2	85.4 (0.9)	84.5 (1.9)	82.0 (4.8)	78.0 (9.5)	70.3 (18.4)
TnT	88.9	88.1 (0.9)	87.3 (1.8)	85.2 (4.2)	81.7 (8.1)	74.9 (15.7)

Table 8.2: *Accuracy in percent from the parsing task. Parsing based on the* Corpus *tagger had 88.4% accuracy. A baseline parser using the* Corpus *tagger had 59.0% accuracy.*

For the parsing task, we obtained 86.2% accuracy using Brill and 88.9% accuracy using TnT, as seen in Table 8.2. An interesting observation is that the accuracy of parsing using Corpus, i.e. perfect tagging, was 88.4%, which is lower than that of TnT. The explanation is found in the way the taggers based on statistics generalize from the training data. The Corpus tagger adopts the noise from the manual annotation of the SUC corpus, which will make the task harder for the parser. This is further substantiated below when we discuss the baseline parser.

The degradation at the 20% error level seems promising since the accuracy only dropped 15.7% using the TnT tagger. Since the performance of TnT had already degraded 13.1% in tagging accuracy, the additional $15.7 - 13.1 = 2.6\%$ was due to the fact that the context surrounding a tagging error was erroneously parsed. This difference is the degradation of the parser in isolation. Nevertheless, the performance of the whole system is the most relevant measure, since the most accurate tagger does not necessarily provide the best input to the rest of the parsing system.

As a comparison, the baseline parser using the Corpus tagger had 59.0% accuracy, while the TnT tagger obtained 59.2%. This further indicates that the difference between TnT and Corpus is real and not just an idiosyncrasy of the parsing system. A system not using any knowledge at all, i.e. the baseline parser using the Base tagger, obtained 55.5% accuracy.

As seen from Table 8.3, the task of clause identification (`CLB`) was more robust to ill-formed input than any other task with only 7.0% degradation using TnT at the 20% error level. This may be attributed to the fact that half the clause delimiters resided at the beginning of a sentence and thus, were unaffected by spelling errors. Clearly, the baseline clause identifier was also unaffected by spelling errors and obtained a 69.0% F-score for all error levels. Clause identification at the 0% error level achieved an 88.3% F-score (88.3% recall, 88.3% precision) using TnT. Using the Corpus tagger, we achieved 88.2%, which is once again lower than TnT.

Tagger	0%	1%	2%	5%	10%	20%
Base	84.2	84.0 (0.2)	83.6 (0.7)	82.9 (1.5)	81.9 (2.7)	79.4 (5.7)
Brill	87.3	87.0 (0.3)	86.6 (0.8)	85.6 (1.9)	83.8 (4.0)	80.3 (8.0)
TnT	88.3	87.9 (0.4)	87.5 (0.9)	86.6 (1.9)	85.1 (3.6)	82.1 (7.0)

Table 8.3: *F-score from the clause boundary identification task. Identification based on the* Corpus *tagger had an F-score of 88.2%. A baseline identifier had an F-score of 69.0%. The columns correspond to the percentage of errors introduced. Relative accuracy degradation compared to the 0% error level is given in brackets.*

We provide the F-scores for the individual phrase categories using TnT and Brill in Tables 8.4 and 8.5. In the count column, the number of rows in which a given phrase type occurs in the annotation are given. For example, in the case of NP, we count the number of rows in which at least one `NPB` or `NPI` occurs in the treebank.

For TnT, we see that adverbial (ADVP) and infinitival verb phrases (INFP) are much less accurate than the others. They are also among the most sensitive to ill-formed input. In the case of INFP, this may be attributed to the fact that they are often quite long and an error introduced near or at the infinite marker or the verb is detrimental. The adjective phrases (AP) have the highest degradation

Type	0%	1%	2%	5%	10%	20%	Count
ADVP	81.9	81.3 (0.7)	80.6 (1.5)	78.6 (4.0)	75.3 (8.0)	68.4 (16.4)	1008
AP	91.3	90.5 (0.8)	89.8 (1.6)	87.0 (4.7)	83.1 (8.9)	74.3 (18.6)	1332
INFP	81.9	81.4 (0.6)	80.9 (1.2)	79.2 (3.2)	76.0 (7.2)	70.2 (14.2)	512
NP	91.4	90.9 (0.5)	90.2 (1.3)	88.4 (3.2)	85.2 (6.7)	79.3 (13.2)	6895
O	94.4	94.2 (0.2)	93.9 (0.5)	93.3 (1.1)	92.1 (2.4)	89.9 (4.7)	2449
PP	95.3	94.8 (0.5)	94.3 (1.0)	93.0 (2.4)	90.9 (4.6)	85.8 (9.9)	3886
VC	92.9	92.3 (0.6)	91.5 (1.5)	89.8 (3.3)	86.8 (6.5)	80.9 (12.9)	2562
Total	88.9	88.1 (0.9)	87.3 (1.8)	85.2 (4.2)	81.7 (8.1)	74.9 (15.7)	

Table 8.4: *F-scores for the individual phrase categories from the parse task using the* TnT *tagger.*

Type	0%	1%	2%	5%	10%	20%	Count
ADVP	80.6	80.1 (0.6)	79.3 (1.6)	77.9 (3.3)	74.4 (7.6)	67.6 (16.1)	1008
AP	87.7	86.8 (1.0)	85.8 (2.1)	82.4 (6.0)	77.9 (11.1)	68.5 (21.8)	1332
INFP	80.8	80.4 (0.4)	79.2 (1.9)	77.9 (3.5)	73.6 (8.9)	67.7 (16.2)	512
NP	88.8	88.1 (0.7)	87.2 (1.8)	84.8 (4.5)	80.9 (8.8)	73.9 (16.7)	6895
O	93.8	93.5 (0.3)	93.2 (0.6)	92.5 (1.3)	91.0 (2.9)	88.4 (5.7)	2449
PP	93.4	92.9 (0.5)	92.3 (1.1)	90.7 (2.8)	88.1 (5.6)	82.5 (11.6)	3886
VC	90.9	90.2 (0.7)	89.3 (1.7)	87.1 (4.1)	83.4 (8.2)	75.8 (16.6)	2562
Total	86.2	85.4 (0.9)	84.5 (1.9)	82.0 (4.8)	78.0 (9.5)	70.3 (18.4)	

Table 8.5: *F-scores for the individual phrase categories from the parse task using the* Brill *tagger.*

of all. An AP is always part of an NP and thus, it will be difficult to parse if either the adjective is misspelled or the NP is disturbed. We see that high accuracy and robustness for the tagger yield high accuracy and robustness for the phrase recognition as TnT always has higher accuracy on error-free text than Brill and always less degradation (with one exception: the degradation of ADVP, which is too small to draw any conclusions).

Standard deviation was calculated for all accuracy and F-score values at each error level, by using data from all ten files from a specific error level. Standard deviations were low for all tasks and were 0.13, 0.22 and 0.22 on the average for Tables 8.1, 8.2 and 8.3, respectively. The maximum standard deviation using TnT was 0.70 for the 20% error level for clause boundary identification. The standard deviation was 0.49 on the average for Tables 8.4 and 8.5. The only noticeable exception was the infinitival verb phrase (INFP), which had a 2.5 standard deviation at the 20% error level using the Brill tagger.

8.6 Spelling Error Correction

As stated in Section 7.2, we use spelling errors to simulate noise in text. The use of a spell checker to correct the spelling errors would greatly affect the input to the parser and thus, would affect the results. However, it is not clear that the correction of spelling errors would improve the performance of the parser. In Chapter 10, we evaluated three spell checkers for Swedish. The best results for spelling error correction were obtained by STAVA (Kann et al., 2001). Using the first suggestion from STAVA we would correctly change about 85% of the misspelled words into the correct word. However, the remaining 15% of the misspelled words would be changed into another, unrelated word. The introduction of unrelated words is certainly problematic for the tagger and parser.

Spelling errors	1%	2%	5%	10%	20%
Auto-corrected	87.9	87.1	84.4	80.2	72.4
Not auto-corrected	88.1	87.3	85.2	81.7	74.9

Table 8.6: *Accuracy in percent for the GTA parser. The first row contains the results when the spelling errors were automatically corrected by the* STAVA *spell checker. The second row contains the results when the misspelled words were not corrected.*

We used the GTA parser to determine the effect of applying an automatic spelling corrector. The same 50 misspelled files were used as in the evaluation above. The results are shown in Table 8.6. We see that parsing text while retaining the errors obtains higher accuracy than parsing after having corrected the errors, for all error levels. Evidently, the 15% words that are changed into an unrelated word make the processing difficult since the tagger's inherent robustness to misspelled words cannot be used.

8.7 Discussion

We have evaluated the GTA parser on 14 000 words. However, we realize that this may not be sufficient for a reliable conclusion on robustness for the GTA parser. The experiments here are primarily provided to illustrate the evaluation method. Nevertheless, the results show that the GTA parser applied on TNT or BRILL output degrades less than the amount of errors introduced. Furthermore, the taggers are very robust to noisy input as TNT degrades only 13.1% and BRILL degrades 15.2% at the 20% error level. The parser itself adds only a few per cent units of degradation (about 3% for both taggers) at the 20% error level. This leads us to believe that the parser itself is quite robust and that the most critical part of a robust parser is the robustness of the part-of-speech tagger.

We noted that the TnT tagger actually achieved a higher parser accuracy than the Corpus tagger. That is, the information learned by the TnT tagger was more useful than the information used for training, which is quite surprising. The same behavior was also observed for the clause identification task. However, while the Corpus tags are more accurate, the second-order Markov model learns general patterns and thus, hides some of the idiosyncrasies of the corpus annotations. Furthermore, the repeated process of parser rule refinement is carried out on the parser output from a PoS tagger and not the original tags of the corpus. Evidently, this will favor the patterns learned by the PoS tagger.

As seen from the section on related work, many approaches have been proposed to evaluate robustness against noisy data. However, they all required a treebank containing noisy text. Missplel is capable of introducing almost any type of error produced by a human. Hence, we can simulate human errors in text to almost any detail. By using the proposed method, we have obtained detailed information about the robustness of the parser and its components without any requirements of a resource annotated with errors. We see that the proposed method provides accurate measurements of robustness and avoids the problems with different text genres and extraneous manual work. Nevertheless, in the next chapter, we propose an unsupervised equivalent to the method proposed here, totally eliminating the need for annotated resources and manual work.

Chapter 9

Unsupervised Evaluation of Parser Robustness

The evaluation in the previous chapter was accurate and free from manual labor, assuming the existence of an annotated resource. However, if the proper resources do not exist, manual labor is required to produce such a resource. Furthermore, existing resources will be obsolete if the parser output is changed or upgraded to include more detailed analysis. This chapter presents a fully unsupervised evaluation method for parser robustness. Thus, the proposed method totally eliminates the need for manual labor and annotated resources, but still provides accurate figures on parser robustness.

The only requirements of the evaluation method are a (relatively error-free) text and an estimate of the accuracy of the parser (on error-free text, which is usually known). Despite the modest requirements, the evaluation procedure provides accurate estimates of the robustness of an NLP system, as shown by an evaluation of the proposed method.

9.1 Automation and Unsupervision

The evaluation procedure proposed here is unsupervised and does not require any type of annotated resource. Nevertheless, to assess the accuracy of the evaluation, we used several annotated resources.

9.2 Related Work

We have not been able to find any references to unsupervised evaluation of parser robustness in the literature. The available robustness evaluations focus on the use or manipulation of existing resources. To name a few examples, Basili and Zanzotto (2002) divide Italian and English treebanks into levels of difficulty based on the number of syntactic dependencies. From that, they evaluate the robustness against

increasing levels of language complexity. Vilares et al. (2004) use only a small subset of English and use a different definition of robustness than ours (one more focused on efficient processing of large amounts of trees). Also, in the Related Work section of the previous chapter, we reported on a supervised evaluation scheme by Li and Roth (2001) and a supervised evaluation involving manual work by Foster (2004). The proposed method requires no annotated resources of any kind. It provides a time-saving framework for evaluating parser robustness and since any text may be used, the evaluation method is language independent. Furthermore, the estimates from the procedure were accurate, as seen from a supervised evaluation of the proposed method.

9.3 Proposed Method

We are given an NLP system processing and outputting row-based data, that is, reading one word per row and producing one output (e.g. a parse string) per row. We want to assess the robustness of the system. To this end, we need to evaluate the performance of the system when applied to input with increasing amounts of noise. The proposed method is applicable to most NLP system, but parsers will be used here to provide an example of the evaluation procedure.

Naturally, the performance of an NLP system can be better assessed with an annotated resource. To begin with, the discussion here will include such a resource. The aim is to establish how much information can be gained concerning the performance of the NLP system *without* the annotated resource.

We require a text to be used in the evaluation. The text will be processed by the NLP system (i.e. a parser). Even though the text can be chosen arbitrarily, we simplify the exposition of the method by using the text from the annotated resource mentioned previously; but keep in mind that the method does not require an annotated resource. We introduce spelling errors in the text to determine the performance of the NLP system under the influence of noisy and ill-formed input, as described in Section 7.2. Thus, we use Missplel to introduce spelling errors simulating keyboard mistypes. To avoid alternate interpretations of a sentence, the spelling errors result only in words not present in a dictionary. The reasons for choosing spelling errors to simulate noisy input are given in Section 7.2.

Three different data sources are involved in the discussion of the evaluation method. The three files have the same number of rows since they all originate from the same text (i.e. the text in the treebank). For each row, they contain a word (that may or may not be misspelled) and a parse string for that word. Only the parse part is used here.

The first file, denoted `m`, is the manually checked annotated resource (e.g. a tree bank). The second file, denoted `0` (zero), is the output of the NLP system when applied to the original treebank text (0% errors). The third file, denoted `n`, is the output of the NLP system when applied to the text containing errors (e.g. $n = 5\%$

of the words in the file are misspelled). Clearly, a file containing $n\%$ errors is more difficult to parse than an error-free text and we want to determine how difficult.

Five Cases

Given one row of the treebank, the 0% file and the $n\%$ file, we analyze the different cases that may occur. Say that the treebank parse (i.e. the correct answer) is `a`. The 0% file either contains the correct answer `a`, or an incorrect answer `b`. Furthermore, the $n\%$ file may contain the correct answer `a`, the same incorrect answer `b` as the 0% file or even another incorrect answer `c`. From this, we obtain several different combinations.

We introduce a notation (denoted `m0n`) consisting of three columns. The first position is the parse found in the treebank `m`, the second is the 0% file `0` and the third is the $n\%$ file `n`. For example, `abc` means that the parse from the treebank was `a`, the parse from the 0% file was `b` and the parse found in the $n\%$ file was `c`.

Thus, using the new notation, we get five different cases when comparing parses of a single word: `aaa`, `aab`, `aba`, `abb` and `abc`, as shown in Table 9.1. The first case `aaa` is the most common, where all three files agree on the same parse. Second, `aab` is the case where an error nearby in the text corrupted the parsing process of this row. The third case `aba` is unusual, but not negligibly so. This may occur when the parser is uncertain and chooses between two equal alternatives and arbitrarily chooses the correct one at the $n\%$ level due to a nearby error in the text. The fourth case `abb` is common and occurs when the parser does not know how to parse a correct grammatical construction. The last case `abc` may be caused by an error introduced near a correct grammatical construction that the parser cannot parse correctly. This case is uncommon. See Table 9.2 for an example of the five cases.

x	m0n	m = 0?	m = n?	0 = n?	x (5%)	x (10%)
x_{aaa}	aaa	=	=	=	85%	81%
x_{aab}	aab	=			4.0%	7.8%
x_{aba}	aba		=		0.32%	0.64%
x_{abb}	abb			=	10%	9.1%
x_{abc}	abc				0.77%	1.4%

Table 9.1: *An example of the relative frequencies of the five cases with 5% and 10% errors in the text (for the GTA parser from Section 5).*

Let $x_{aaa}, x_{aab}, x_{aba}, x_{abb}$ and x_{abc} correspond to the relative frequencies of the five cases in Table 9.1. For example, if `abb` occupies 10% of the rows, $x_{abb} = 0.10$. Clearly,

$$x_{aaa} + x_{aab} + x_{aba} + x_{abb} + x_{abc} = 1, \tag{9.1}$$

(treebank) word	manual annotation	(error-free text) word	parser output	(n% errors) word	parser output	case
Vi	NPB	Vi	NPB	Vi	NPB	`aaa`
kan	VPB	kan	VPB	*kna*	VPB	`aaa`
välja	VPI	välja	VPI	välja	**VPB**	`aab`
att	NPB	att	**O**	att	NPB	`aba`
säga	VPB|NPI	säga	**VPB**	säga	VPB|NPI	`aba`
upp	VPI|NPI	upp	**VPI**	*upö*	**NPB|NPI**	`abc`
avtalet	NPB|NPI	avtalet	**NPB**	avtalet	**NPB**	`abb`

Table 9.2: *Examples of the five cases resulting from parsing a single word. Translation: Vi (We) kan (can) välja (choose) att (to) säga upp (cancel) avtalet (the agreement). Explanation of the GTA parser output is given in Section 5.5.*

since they cover all possible outcomes. Let acr_{m0} denote the accuracy when comparing the `m` column to the `0` column. We see that

$$acr_{m0} = x_{aaa} + x_{aab} \tag{9.2}$$

since only in cases `aaa` and `aab`, the two columns `m` and `0` both contain the same output `a`. Furthermore, by the same reasoning,

$$acr_{mn} = x_{aaa} + x_{aba} \quad \text{and} \tag{9.3}$$

$$acr_{0n} = x_{aaa} + x_{abb}. \tag{9.4}$$

The x_{abb} is included in the last equality since `0` equals `n` in `abb` even though they both differ from `m`. The fact that they differ from the treebank cannot be established without the correct answer `m`.

We say that the performance of the NLP system *degrades* when the performance decreases with increasing levels of errors in the text. The degradation $degr_n$ is a comparison between the performance at the n% error level and the performance at the 0% error level. Let

$$degr_n = 1 - \frac{acr_{mn}}{acr_{m0}}. \tag{9.5}$$

Clearly, this is calculable only if you have access to acr_{mn} and acr_{m0}.

Normally, some sort of evaluation has been carried out to estimate the accuracy of the parser on error-free text, denoted acr. High accuracy is obtained when the correct answer `m` often corresponds to the output `0`. Thus, the accuracy is a very good estimate for acr_{m0} and we will use $acr_{m0} = acr$. Nevertheless, without the annotated resource, we do not have access to or estimates for acr_{mn}.

Upper and Lower Bounds

We want to estimate the degradation $degr_n$ without knowing acr_{mn}. Without the annotated resource, we only have access to acr_{0n} and $acr_{m0} = acr$. We will use these to establish an upper bound $degr_n^{upr}$ for $degr_n$. We want the value $degr_n^{upr}$ to be an arbitrary expression including acr and acr_{0n} that can be proven to be greater than $degr_n$. We propose

$$degr_n^{upr} = \frac{1 - acr_{0n}}{acr} \tag{9.6}$$

as an upper bound. We prove that $degr_n^{upr}$ is always greater than $degr_n$ by letting

$$degr_n^{upr} = degr_n + \epsilon. \tag{9.7}$$

Equations (9.1)–(9.2) and (9.4)–(9.6) give us

$$\epsilon = \frac{2x_{aba} + x_{abc}}{acr}. \tag{9.8}$$

We see that $\epsilon \geq 0$ since all $x \geq 0$ and thus, $degr_n^{upr} \geq degr_n$ as required.

The smaller the value of ϵ, the better. From the discussion above, we saw that x_{aba} and x_{abc} are normally quite small, which is promising.

We now turn to a lower bound for $degr_n$. Similar to the upper bound, the lower bound can be an arbitrary expression containing acr_{0n} and acr. We propose

$$degr_n^{lwr} = \frac{1}{2} degr_n^{upr} = \frac{1 - acr_{0n}}{2acr}. \tag{9.9}$$

Again, as for the upper bound, the expression must be proven to be less than $degr_n$. To this end, we let

$$degr_n^{lwr} + \delta = degr_n. \tag{9.10}$$

From Equations (9.1)–(9.2), (9.4)–(9.5) and (9.9)–(9.10), we obtain

$$\delta = \frac{x_{aab} - 3x_{aba} - x_{abc}}{2acr}, \tag{9.11}$$

which is non-negative when $x_{aab} \geq 3x_{aba} + x_{abc}$.

Both cases `aab`, `aba` and `abc` are the result of an introduced spelling error. With no errors, x_{aab}, x_{aba} and x_{abc} are all zero and with increased levels of introduced errors, they will all increase. Hence, x_{aab}, x_{aba} and x_{abc} are positively correlated. Furthermore, it is clear that case `aab` is much more common than `aba` and `abc` since it involves correctly parsed text at the 0% error level. The accuracy acr determines the amount of correctly parsed text and thus, with reasonable accuracy, the above inequality holds with a good margin of error. See Section 9.7 for details on the conditions under which the above inequality holds. Section 9.4 further supports that the inequality holds, since in all experiments is the left-hand side more than

twice the right-hand side. Using Table 9.1 as an example, the right-hand side of the inequality is 1.73% for the 5% column which is less than half of $x_{aab} = 4.0\%$. For the 10% column, the right-hand side is 3.32%, which is less than half of $x_{aab} = 7.8\%$.

From the above discussion and given the conditions, we have obtained

$$degr_n^{lwr} \leq degr_n \leq degr_n^{upr}. \tag{9.12}$$

Estimation of the Degradation

The simple relationship between the upper and lower bounds allows us to deduce some further information. Given an upper bound $degr_n^{upr}$ and a lower bound $degr_n^{lwr}$, we want to estimate the position of the true value $degr_n$. Clearly, $degr_n$ is somewhere in between $degr_n^{lwr}$ and $degr_n^{upr}$ from Equation (9.12). Let $degr_n^{est}$ be the center of the interval contained by the lower and upper bound, that is,

$$degr_n^{est} = \frac{1}{2}(degr_n^{lwr} + degr_n^{upr}) \tag{9.13}$$

and let γ be the distance from $degr_n$ to $degr_n^{est}$. Then,

$$degr_n + \gamma = degr_n^{est}. \tag{9.14}$$

Equations (9.7), (9.10) and (9.13) yield $\gamma = (\epsilon - \delta)/2$. Using Equations (9.8) and (9.11) results in the explicit form

$$\gamma = \frac{7x_{aba} + 3x_{abc} - x_{aab}}{4acr}. \tag{9.15}$$

We see that γ is small if $7x_{aba} + 3x_{abc} \approx x_{aab}$. If we use Table 9.1 as an example, we obtain $\gamma = 0.0015$ for the 5% error level and $\gamma = 0.0025$ for the 10% error level given that the accuracy acr is 89% for the GTA parser. Thus, for these particular examples, the estimate $degr_n^{est}$ differs from the real degradation $degr_n$ by no more than a quarter of a per cent unit!

As the discussion above about the lower bound illustrated, x_{aab}, x_{aba} and x_{abc} are correlated, which is promising if γ is to be small for all error levels simultaneously. See Section 9.7 for a discussion on the conditions required to make γ small. Though the experiments in Section 9.4 show that γ is quite small, we make no claims that γ is equally small for all NLP systems. The estimations here are just theoretical indications where the true value of $degr_n$ may reside.

We have indicated that $degr_n^{est}$ is, in theory, close to $degr_n$. By using Equations (9.6) and (9.9), we simplify and obtain an explicit formula for the estimated degradation:

$$degr_n^{est} = \frac{3}{4} degr_n^{upr} = \frac{3(1 - acr_{0n})}{4acr}. \tag{9.16}$$

Hence, without having an annotated resource, we can estimate the robustness (degradation) of the system quite accurately.

Accuracy

Now that the degradation of the performance has been established, we turn to the accuracy. The definition of $degr_n$ in Equation (9.5) states that $degr_n = 1 - acr_{mn}/acr$. We are interested in the accuracy of the NLP system on the $n\%$ file, that is, acr_{mn}. Rearranging the above equation yields

$$acr_{mn} = acr(1 - degr_n). \tag{9.17}$$

Since $degr_n$ is unknown, we use $degr_n^{upr}$, $degr_n^{lwr}$ and $degr_n^{est}$ to obtain bounds on the accuracy:

$$\begin{aligned} acr_{mn}^{lwr} &= acr(1 - degr_n^{upr}), & (9.18)\\ acr_{mn}^{upr} &= acr(1 - degr_n^{lwr}), & (9.19)\\ acr_{mn}^{est} &= acr(1 - degr_n^{est}). & (9.20) \end{aligned}$$

The estimation in Equation (9.20) is not precise, so we let

$$acr_{mn} + \lambda = acr_{mn}^{est}. \tag{9.21}$$

From Equations (9.14), (9.17) and (9.20), we obtain

$$\lambda = acr \cdot (-\gamma). \tag{9.22}$$

Thus, if $|\gamma|$ is small, $|\lambda|$ is even smaller, and thus, acr_{mn}^{est} is a good approximation of the accuracy of the NLP system when applied to a file containing $n\%$ errors.

To summarize, the theory of the evaluation procedure is presented in Table 9.3.

9.4 Experiments

Five different parsers were used to assess the accuracy of the evaluation method.

GTA from Section 5 is a rule-based shallow parser. It relies on hand-crafted rules of which a few are context-sensitive. The rules are applied to part-of-speech tagged text. GTA identifies constituents and assigns phrase labels but does not build full trees with a top node. Example output from the GTA parser is given in Figure 9.1.

FDG (Voutilainen, 2001), Functional Dependency Grammar, is a commercial dependency parser. It builds a connected tree structure, where every word points at a dominating word. Dependency links are assigned a function label. FDG produces other information too, such as morphological analysis and lemma of words, which is not used here. Example output from the FDG parser is given in Figure 9.2.

The dependency parser by Nivre (2003) uses a manually constructed grammar and assigns dependency links between words, working from part-of-speech tagged text. We denoted it the MCD parser (manually constructed dependency). Example output from the MCD parser is given in Figure 9.3.

acr	**required:** estimated accuracy of the NLP system on error-free text
$acr_{m0} \approx acr$	**assumption:** system accuracy on test text is close to acr
acr_{0n}	**known:** obtainable without annotated resource
acr_{mn}	**unknown:** accuracy of the NLP system on erroneous text
$degr_n = 1 - acr_{mn}/acr_{m0}$	**sought:** degradation (robustness) of the NLP system
$degr_n^{upr} = (1 - acr_{0n})/acr$	upper bound for degradation
$degr_n \leq degr_n^{upr}$	degradation is bounded from above
$degr_n^{est} = \frac{3}{4} degr_n^{upr}$	approximation of degradation
$degr_n^{est} - degr_n = \gamma$	approximation is not exact
$\gamma = (7x_{aba} + 3x_{abc} - x_{aab})/4acr$	deviation of the approximation
$acr_{mn}^{lwr} = acr(1 - degr_n^{upr})$	lower bound for accuracy on erroneous text
$acr_{mn}^{lwr} \leq acr_{mn}$	accuracy is bounded from below
$acr_{mn}^{est} = acr(1 - degr_n^{est})$	approximation of accuracy
$acr_{mn}^{est} - acr_{mn} = \lambda$	approximation is not exact
$\lambda = acr \cdot (-\gamma)$	deviation of the approximation
if $x_{aab} \geq 3x_{aba} + x_{abc}$	**condition:** required for the lower bound on the degradation
$degr_n^{lwr} = \frac{1}{2} degr_n^{upr}$	lower bound for degradation
$degr_n^{lwr} \leq degr_n \leq degr_n^{upr}$	degradation is bounded if condition is met
$acr_{mn}^{upr} = acr(1 - degr_n^{lwr})$	upper bound for accuracy on erroneous text
$acr_{mn}^{lwr} \leq acr_{mn} \leq acr_{mn}^{upr}$	accuracy is bounded if condition is met

Table 9.3: *Summary of the theory of the evaluation procedure.*

The Malt parser (Nivre et al., 2004), another dependency parser, is based on the same algorithm as MCD but uses a memory-based classifier trained on a treebank instead of a manually constructed grammar. Unlike MCD, the Malt parser not only assigns dependency links between words but also attaches function labels to these links. Example output from the Malt parser is given in Figure 9.4.

A manually constructed context-free grammar for Swedish was used with an implementation of Earley's parsing algorithm, as described in (Megyesi, 2002a). We denoted it the Earley parser. Example output from the Earley parser is given in Figure 9.5.

The GTA, MCD, Malt and Earley parsers are all under development. All parsers had row-based output, that is, one word and one parser output per row. The GTA and Earley parsers used the IOB format, explained in Section 5.5. However, they do not produce the same analysis, so the results are not directly comparable. Malt and MCD are similar in their construction but their results are not really comparable

```
Dekoren      NPB
har          VCB
stiliserats  VCI
och          0
förenklats   VCB
.            0
```

Figure 9.1: *Output example from the GTA parser.*

```
Dekoren      subj:>75640
har          v-ch:>75641
stiliserats  main:>75641
och          cc:>75641
förenklats   cc:>75641
.            .
```

Figure 9.2: *Output example from the FDG parser.*

```
Dekoren      1
har          0
stiliserats  -1
och          0
förenklats   -3
.            0
```

Figure 9.3: *Output example from the MCD parser.*

since Malt assigns function labels and MCD does not. On unlabeled output, Malt is more accurate than MCD.

The output from the TnT tagger was used as input for all parsers but FDG, which includes its own tagger. Example output from the TnT tagger is given in Figure 9.6.

Parser Robustness Evaluation

In the evaluation, we used 100 000 words from the Stockholm-Umeå Corpus (SUC), described in Section 7.1. The 100 000 word text was parsed using each of the

```
Dekoren      2,SUB
har          3,VC
stiliserats  5,SUB
och          5,SUB
förenklats   0,ROOT
.            5,IP
```

Figure 9.4: *Output example from the Malt parser.*

```
Dekoren      NPB
har          VCB
stiliserats  VCI
och          0
förenklats   VCB
.            0
```

Figure 9.5: *Output example from the Earley parser.*

```
Dekoren      nn.utr.sin.def.nom
har          vb.prs.akt.aux
stiliserats  vb.sup.sfo
och          kn
förenklats   vb.sup.sfo
.            mad
```

Figure 9.6: *Output example from the* TnT *tagger. The tag set is explained in Table 7.1.*

parsers. The parse results of this error-free text (0% errors) constituted the `0` file, as defined in the first part of Section 9.3. Spelling errors (resulting in non-existing words only) were randomly inserted into the text using Missplel, as described in Sections 7.2 and 9.3. The parse results from the misspelled text (containing e.g. 5% errors) constituted the `n` file, also from Section 9.3. For the GTA, the MCD and the Malt parser and the TnT tagger, manually annotated resources were available. The experiments on these resources are reported in the next section.

To see how the parser behaves with increasing amounts of errors, $n = 1\%$, 2%, 5%, 10% and 20% of all words were randomly misspelled. To reduce the influence of

chance, 10 different misspelled files were created for each error level. Using these, we calculated the mean for the degradation, the accuracy and so forth. To simplify the evaluation, AutoEval (from Chapter 3) was used for input and output handling and data processing. The variance between different files was low.

The degradation estimates for a particular file were obtained by calculating acr_{0n}, that is, by comparing how many of the parses in the `0` file that corresponded to the parses in the `n` file. From acr_{0n} we calculated the upper and lower bounds as well as estimates on the degradation and accuracy, as seen in the summary in Table 9.3.

In the experiments, any deviation from the correct parse was considered an error, even if it was "almost" correct (though the evaluation method could just as easily use a more sophisticated analysis). Hence, parsers that provide richer information will generally be less robust than parsers that return less information, since there are more possibilities for errors.

Comparing the output of FDG on different versions of the same text is non-trivial, since the tokenization may be altered by a misspelled word. Here, any tokens without a directly corresponding token in the other text were ignored. All other tokenization difficulties were interpreted to give FDG as many "correct" parses as possible. The 90% accuracy for FDG is our estimation.

Evaluating the Evaluation Method

Due to the kind contribution of the parser implementers, text with correctly annotated parse output was available for some of the parsers, though only in small amounts. By using these, we wanted to assess the accuracy of the proposed method.

For the GTA parser and the TnT part-of-speech tagger, we had a 14 000 word file of manually corrected parse and tag data, as described in Section 7.1. For the MCD parser, we had a 4 000 word file and for Malt we had 10 000 words. We used the text from the annotated files and carried out the same procedure as in the previous subsection, that is, introduced errors and evaluated. We also had the correct answers from the annotated resource. From this, we calculated the real degradation and accuracy as shown in the next section.

9.5 Results

The results for the five parsers are presented in Tables 9.4 through 9.8, which also present the accuracy *acr* on error-free text. The first column reports on the amount of errors in the text. The second column is the amount of parse output that differs between the rows of the `0` file and the `n` file. This value is $1 - acr_{0n}$. The third column presents the degradation of the parser. The first value is the lower bound $degr_n^{lwr}$ and the second is the upper bound $degr_n^{upr}$. The figure in parentheses is the estimated degradation $degr_n^{est}$. The fourth column contains the estimations on the accuracy: lower bound acr_{mn}^{lwr}, upper bound acr_{mn}^{upr} and estimated value acr_{mn}^{est}.

Error level	Parse differs	Estimated degradation	Estimated accuracy
1	1.2	0.7 – 1.3 (1.0)	88 – 88 (88)
2	2.4	1.3 – 2.6 (2.0)	87 – 88 (87)
5	5.7	3.2 – 6.4 (4.8)	83 – 86 (85)
10	11	6.2 – 12 (9.4)	78 – 83 (81)
20	21	12 – 24 (18)	68 – 78 (73)

Table 9.4: *Estimated robustness of the GTA parser on 100 000 words. All figures are given in per cent. Estimated accuracy on error-free text: 89%.*

Error level	Parse differs	Estimated degradation	Estimated accuracy
1	0.9	0.5 – 1.1 (0.8)	81 – 82 (82)
2	1.7	1.1 – 2.1 (1.6)	81 – 81 (81)
5	4.3	2.6 – 5.3 (4.0)	78 – 80 (79)
10	8.6	5.2 – 10 (7.8)	74 – 78 (76)
20	17	10 – 20 (15)	66 – 74 (72)

Table 9.5: *Estimated robustness of the MCD parser on 100 000 words. All figures are given in per cent. Estimated accuracy on error-free text: 82%.*

Error level	Parse differs	Estimated degradation	Estimated accuracy
1	1.8	1.2 – 2.4 (1.8)	77 – 78 (77)
2	3.7	2.3 – 4.7 (3.5)	75 – 77 (76)
5	8.9	5.7 – 11 (8.5)	70 – 74 (72)
10	17	11 – 22 (16)	61 – 70 (66)
20	31	20 – 39 (29)	48 – 63 (55)

Table 9.6: *Estimated robustness of the Malt parser on 100 000 words. All figures are given in per cent. Estimated accuracy on error-free text: 79%.*

Error level	Parse differs	Estimated degradation	Estimated accuracy
1	0.8	0.5 – 0.9 (0.7)	89 – 90 (89)
2	1.7	0.9 – 1.8 (1.4)	88 – 89 (89)
5	4.1	2.3 – 4.5 (3.4)	86 – 88 (87)
10	8.2	4.5 – 9.1 (6.8)	82 – 86 (84)
20	16	9.1 – 18 (14)	74 – 82 (78)

Table 9.7: *Estimated robustness of the Earley parser on 100 000 words. All figures are given in per cent. Estimated accuracy on error-free text: 90%.*

Error level	Parse differs	Estimated degradation	Estimated accuracy
1	2.1	1.2 – 2.3 (1.7)	88 – 89 (88)
2	4.2	2.3 – 4.6 (3.5)	86 – 88 (87)
5	10	5.5 – 11 (8.3)	80 – 85 (83)
10	19	11 – 21 (16)	71 – 81 (76)
20	34	19 – 37 (28)	56 – 73 (65)

Table 9.8: *Estimated robustness of the FDG parser on 100 000 words. All figures are given in per cent. Estimated accuracy on error-free text: 90%.*

Error level	Tag differs	Estimated degradation	Estimated accuracy
1	0.7	0.4 – 0.7 (0.6)	95 – 96 (95)
2	1.4	0.7 – 1.5 (1.1)	95 – 95 (95)
5	3.6	1.9 – 3.7 (2.8)	92 – 94 (93)
10	7.2	3.7 – 7.5 (5.6)	89 – 92 (91)
20	14	7.5 – 15 (11)	82 – 89 (85)

Table 9.9: *Estimated robustness of the PoS tagger* TnT *on 100 000 words. All figures are given in per cent. Estimated accuracy on error-free text: 96%.*

The proposed method evaluates the robustness on one row at the time. For example, if the first column says 5%, we have introduced errors in 5% of the words (with one word per row). Similarly, if we report 11% in the second column (parse differs), then 11% of the parse output (with one parse per row) is different between the two files `0` and `n`.

Parsers base much of their decisions on the part-of-speech information assigned to a word. Since part-of-speech taggers often guess the correct tag for regularly inflected unknown words, the part-of-speech tagger is responsible for a large part of the robustness. In Table 9.9, the estimated degradation of the part-of-speech tagger TnT (Brants, 2000) is shown. TnT was used for all parsers but FDG, which includes its own tagger.

The results for the evaluation of the evaluation are provided in Tables 9.10 through 9.13. The main focus of interest is the difference between the estimated degradation (in brackets) and the real degradation, both given in bold. This difference is γ, as defined in Equation (9.14). Clearly, the closer the estimated degradation is to the real degradation, the better.

Furthermore, the results from the large, unlabeled resources (Tables 9.4 through 9.9) and the smaller, annotated resources (Tables 9.10 through 9.13) are summarized in graph form in Figures 9.7 though 9.12. The graphs are divided into sections corresponding to error levels. In each section, the left-most bar corresponds to the degradation estimates on the large, unlabeled resource. The other bar (if available)

Error level	Parse differs	Estimated degradation	**Real degr.**	Estimated accuracy	**Real accur.**
1	1.2	0.7 – 1.4 (**1.0**)	**0.9**	88 – 88 (**88**)	**88**
2	2.3	1.3 – 2.6 (**1.9**)	**1.8**	87 – 88 (**87**)	**87**
5	5.1	2.9 – 5.7 (**4.3**)	**4.2**	84 – 86 (**85**)	**85**
10	9.9	5.5 – 11 (**8.3**)	**8.1**	79 – 84 (**81**)	**82**
20	19	10 – 21 (**16**)	**16**	70 – 80 (**75**)	**75**

Table 9.10: *Estimated and actual robustness of the GTA parser on 14 000 words of manually annotated text. All figures are given in per cent. Parser accuracy on error-free text was 89%.*

Error level	Parse differs	Estimated degradation	**Real degr.**	Estimated accuracy	**Real accur.**
1	0.7	0.4 – 0.8 (**0.6**)	**0.6**	82 – 82 (**82**)	**82**
2	1.7	1.0 – 2.0 (**1.5**)	**1.4**	81 – 82 (**81**)	**81**
5	4.0	2.5 – 4.9 (**3.7**)	**3.2**	78 – 80 (**79**)	**80**
10	8.3	5.0 – 10 (**7.6**)	**6.6**	74 – 78 (**76**)	**77**
20	16	9.6 – 19 (**14**)	**13**	67 – 74 (**71**)	**72**

Table 9.11: *Estimated and actual robustness of the MCD parser on 4 000 words of manually annotated text. All figures are given in per cent. Parser accuracy on error-free text was 82%.*

Error level	Parse differs	Estimated degradation	**Real degr.**	Estimated accuracy	**Real accur.**
1	1.8	1.1 – 2.3 (**1.7**)	**1.3**	77 – 78 (**77**)	**78**
2	3.4	2.2 – 4.3 (**3.2**)	**2.4**	75 – 77 (**76**)	**77**
5	8.7	5.5 – 11 (**8.3**)	**6.1**	70 – 74 (**72**)	**74**
10	16	11 – 21 (**16**)	**12**	62 – 70 (**66**)	**69**
20	30	19 – 38 (**29**)	**23**	48 – 64 (**56**)	**60**

Table 9.12: *Estimated and actual robustness of the Malt parser on 10 000 words of manually annotated text. All figures are given in per cent. Parser accuracy on error-free text was 79%.*

corresponds to the degradation estimates on the annotated resource. The star (if available) is the real degradation. In each bar, the upper line is $degr_n^{upr}$, the lower line is $degr_n^{lwr}$ and the center line is $degr_n^{est}$. For comparison, each error level also has a dotted line where the degradation equals the error level.

Error level	Parse differs	Estimated degradation	**Real degr.**	Estimated accuracy	**Real accur.**
1	1.1	0.6 – 1.1 (**0.9**)	**0.9**	95 – 95 (**95**)	**95**
2	1.9	1.0 – 2.0 (**1.5**)	**1.6**	94 – 95 (**94**)	**94**
5	3.9	2.0 – 4.1 (**3.1**)	**3.5**	92 – 94 (**93**)	**92**
10	7.3	3.8 – 7.6 (**5.7**)	**6.7**	88 – 92 (**90**)	**89**
20	14	7.4 – 15 (**11**)	**13**	82 – 89 (**85**)	**83**

Table 9.13: *Estimated and actual robustness of the TnT part-of-speech tagger on 14 000 words of manually annotated text. All figures are given in per cent. Tagger accuracy with no errors inserted was 96%.*

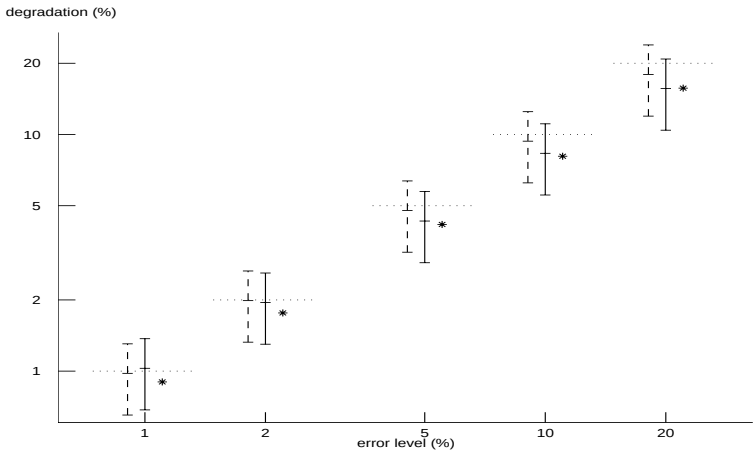

Figure 9.7: *Parser degradation for the GTA parser, log scale.*

9.6 Discussion

From the results, we see that, as guaranteed by the proposed method, the real degradation and accuracy are always between the lower and upper bound. We see that, with few exceptions, the estimated degradation and accuracy are close or equal to the real degradation and accuracy, as indicated in the discussion about γ and λ in Section 9.3. Hence, there is strong reason to believe that the estimations on the 100 000 word files in Section 9.5 are also accurate. Furthermore, by using the results from a small annotated resource (if available), we obtain a good estimate on the relation γ between the real and the estimated degradation for the 100 000 file.

We see that rich information is a liability for at least two of the parsers, FDG and Malt. This is especially clear in the case of Malt, since its output is an extension of

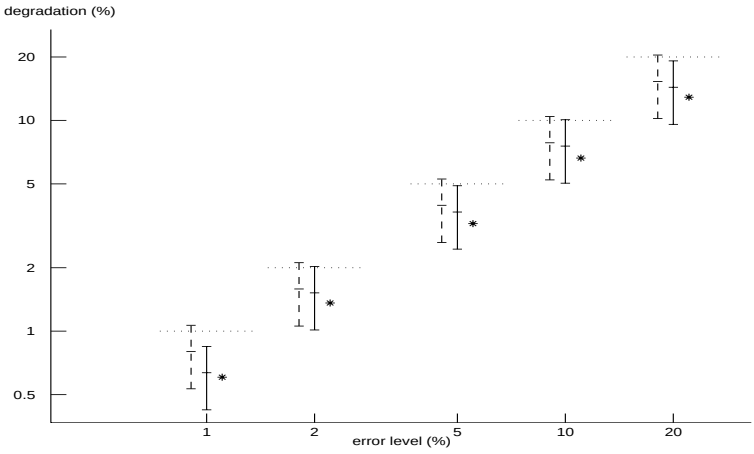

Figure 9.8: *Parser degradation for the MCD parser, log scale.*

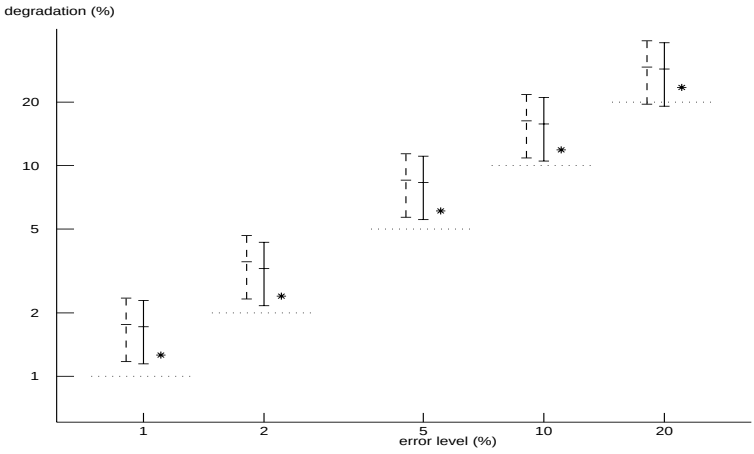

Figure 9.9: *Parser degradation for the Malt parser, log scale.*

that of MCD. The very sparse output of MCD achieves a somewhat higher accuracy and a significantly higher robustness than Malt. Thus, comparing the robustness figures between two parsers is not entirely fair. Nevertheless, if the objective is reluctance to change the output when facing unrestricted and noisy text, the figures are informative.

We note that the proposed method could be used with other types of output

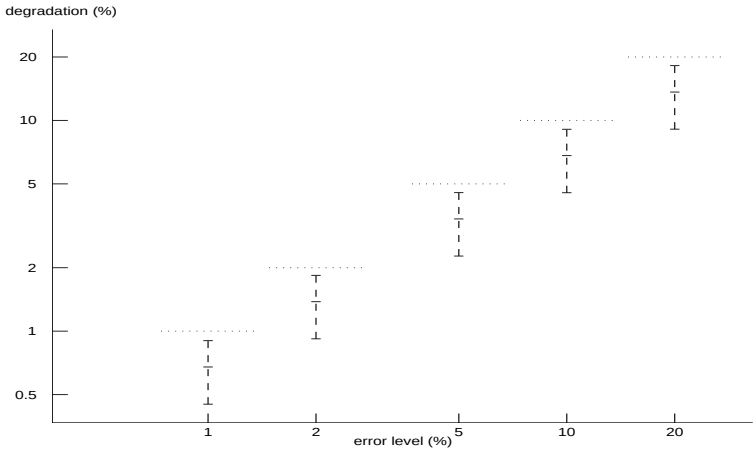

Figure 9.10: *Parser degradation for the Earley parser, log scale.*

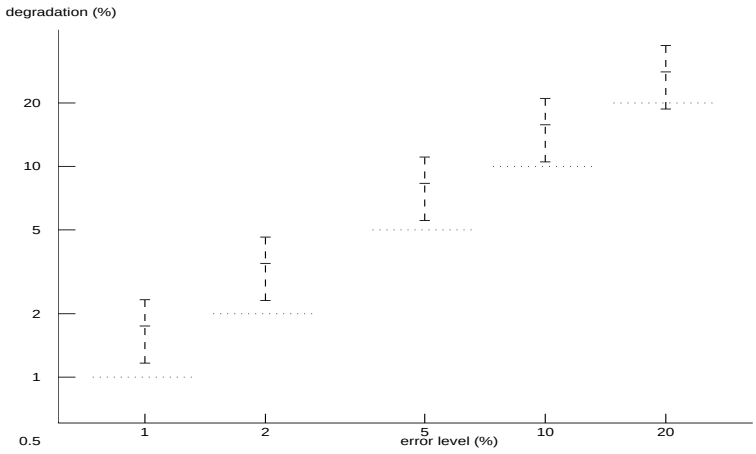

Figure 9.11: *Parser degradation for the FDG parser, log scale.*

besides the row-based used here. Following the guidelines of Lin (1995, 1998), all types of parser information (e.g. bracketed output) could be transformed to row-based data. If we chose not to transform the output, small adjustments may be required of the estimations in the theory section. Also, evaluation of other types of errors would be illustrative of a parser's performance. For example, it would be interesting to evaluate parser robustness on incomplete sentences (in the sense of

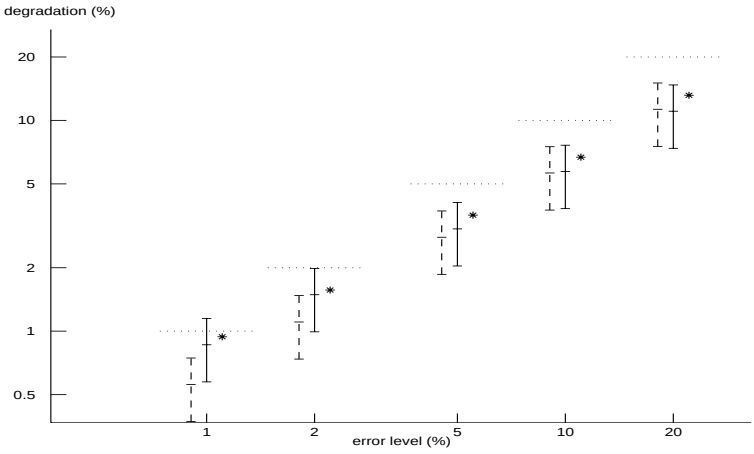

Figure 9.12: *Tagger degradation for the* TnT *tagger, log scale.*

Vilares et al., 2003; Lang, 1988; Saito and Tomita, 1988).

To conclude, we saw that the proposed method required no manual work or annotated resources. Nevertheless, the experiments showed that the evaluation procedure provided very accurate estimates of parser robustness.

9.7 Conditions

In this section, we want to determine the circumstances under which the restriction on δ holds, that is, when

$$\delta = \frac{x_{aab} - 3x_{aba} - x_{abc}}{2acr} \geq 0, \tag{9.23}$$

as discussed in Section 9.3. Furthermore, we will establish the requirements for γ to be small, i.e. when

$$\gamma = \frac{7x_{aba} + 3x_{abc} - x_{aab}}{4acr} \approx 0. \tag{9.24}$$

Assumptions

A few assumptions are required. We know from Equations (9.1) and (9.4) that

$$x_{aab} + x_{aba} + x_{abc} = 1 - acr_{0n}. \tag{9.25}$$

We are interested in an approximation of x_{aab}. To start with, assume that

$$x_{aab}/(1 - acr_{0n}) = acr. \tag{9.26}$$

That is, we assume that x_{aab} compared to the three cases $x_{aab} + x_{aba} + x_{abc}$ is about the same as the accuracy $acr = x_{aaa} + x_{aab}$ compared to one (all cases, $x_{aaa} + x_{aab} + x_{aba} + x_{abb} + x_{abc}$). Or, put another way, the proportion of rows correctly parsed on error-free text (the `m` column compared to the `0` column) should not depend on the number of errors introduced, and thus, it should not matter if looking at all rows or looking only at a sample of the rows. In this case, the sample is the rows affected by a spelling error in the `n` file. We rearrange the assumption in Equation (9.26) and obtain

$$x_{aab} = acr(1 - acr_{0n}). \tag{9.27}$$

We consult Table 9.1 to examine the validity of this estimation. For the 5% column, the right-hand side of Equation (9.27) is 4.45 while x_{aab} is 4.0. For the 10% column, the right-hand side is 8.79 while x_{aab} is 7.8. We see that the values are off by about 12%. This is not very surprising since the sample rows chosen contain the most difficult of all rows (where the perplexity of the parser is the highest) and thus, the accuracy acr_{m0} on these rows is expected to be lower. We introduce a constant $k \leq 1$ denoting the lower accuracy of the sample rows compared to all rows, such that

$$x_{aab}/(1 - acr_{0n}) = k \cdot acr, \tag{9.28}$$

giving us

$$x_{aab} = k \cdot acr(1 - acr_{0n}). \tag{9.29}$$

From Equations (9.25) and (9.29), we get

$$x_{aba} + x_{abc} = (1 - k \cdot acr)(1 - acr_{0n}). \tag{9.30}$$

A discussion on the true value of k is given in the last section of this chapter.

Our second assumption is that

$$x_{aba} \leq x_{abc}. \tag{9.31}$$

The two cases `aba` and `abc` originate from a grammatical construct that could not be parsed by the system. When an error is introduced, the parser changes its output. The most probable is that the change results in something erroneous, as in `abc`.

Results

We use the assumptions in Equations (9.29)–(9.31) with δ in Equation (9.23):

$$\begin{aligned} \delta = \quad & (x_{aab} - 3x_{aba} - x_{abc})/2acr \geq \\ & (x_{aab} - 2(x_{aba} + x_{abc}))/2acr = \\ & (k \cdot acr(1 - acr_{0n}) - 2(1 - k \cdot acr)(1 - acr_{0n}))/2acr \geq 0 \end{aligned}$$

giving us

$$k \cdot acr - 2(1 - k \cdot acr) \geq 0 \qquad \text{and thus,} \qquad acr \geq \frac{2}{3k}. \tag{9.32}$$

Hence, the inequality in Equation (9.23) is satisfied if $acr \geq 2/3 = 67\%$, assuming $k = 1$. We leave the discussion of the true value of k to the next section for the sake of exposition.

We repeat the above process with γ in Equation (9.24) and obtain

$$\begin{aligned} \gamma = \quad & (7x_{aba} + 3x_{abc} - x_{aab})/4acr \geq \\ & (3(x_{aba} + x_{abc}) - x_{aab})/4acr = \\ & (3(1 - k \cdot acr)(1 - acr_{0n}) - k \cdot acr(1 - acr_{0n}))/4acr \geq 0, \end{aligned}$$

giving us

$$3(1 - k \cdot acr) - k \cdot acr \geq 0 \qquad \text{and thus,} \qquad acr \geq \frac{3}{4k}. \tag{9.33}$$

Hence, γ in Equation (9.24) is positive if $acr \leq 3/4 = 75\%$, assuming $k = 1$. On the other hand,

$$\begin{aligned} \gamma = \quad & (7x_{aba} + 3x_{abc} - x_{aab})/4acr \leq \\ & (5(x_{aba} + x_{abc}) - x_{aab})/4acr = \\ & (5(1 - k \cdot acr)(1 - acr_{0n}) - k \cdot acr(1 - acr_{0n}))/4acr \leq 0 \end{aligned}$$

giving us

$$5(1 - k \cdot acr) - k \cdot acr \leq 0 \qquad \text{and thus,} \qquad acr \leq \frac{5}{6k}.$$

Now, γ is negative if $acr \geq 5/6 = 83.3\%$, assuming $k = 1$.

Remarks

The results from the parser evaluation suggest that the value of the constant k is about 0.90 on the average for all parsers. That is, the accuracy on the sample rows is 10% less than the accuracy on all rows. Using $k = 0.90$ in Equations (9.32)–(9.34) we obtain the following results:

$$acr \geq \frac{2}{3k} = 74\% \implies \text{Lower bound is valid} \tag{9.34}$$

$$acr \geq \frac{5}{6k} = 93\% \implies \gamma < 0 \tag{9.35}$$

$$acr \leq \frac{3}{4k} = 83\% \implies \gamma > 0 \tag{9.36}$$

Thus, the value of *acr* where $\gamma = 0$ is, in theory, near

$$(83\% + 93\%)/2 = 88\%. \tag{9.37}$$

These figures could serve as guidelines when assessing the value of γ on NLP system output without an annotated resource.

By looking at the results from the parser evaluation, we can observe the relation between *acr* and γ on authentic data. For the TnT tagger with 96% accuracy, the difference at the 20% error level is $\gamma = 11 - 13 = -2\%$ from Equation (9.14) and Table 9.13. As predicted by the above discussion, NLP systems with high accuracy should have negative γ. The GTA parser has 89% accuracy and obtains $\gamma = 16 - 16 = 0\%$, as seen in Table 9.10. The MCD parser has 82% accuracy and obtains $\gamma = 14-13 = +1\%$ from Table 9.11. For the Malt parser with 79% accuracy, we get $\gamma = 29 - 23 = +6\%$ from Table 9.12. We see that Equations (9.34)–(9.36) are confirmed by these observations since high accuracy yields negative values on γ (e.g. TnT) while lower accuracy yields positive values on γ (e.g. MCD and Malt). Furthermore, accuracy near the center of the interval (88% from Equation (9.37)) gives us a γ close to zero (e.g. GTA with 89% accuracy).

The true value of γ will depend on the quality of the annotations, the difficulty and characteristics of the texts etc. Despite this, the inequalities are useful as guidelines to which parsers will obtain small values for γ. Also, and fortunately, many parsers have an accuracy between 83% and 93%, where the proposed method will accurately predict the degradation without a manual resource.

Chapter 10

Unsupervised Evaluation of Spell Checker Correction Suggestions

This chapter addresses the evaluation of spell checkers. In this context, a spell checker is a piece of software designed to detect a word misspelled into a non-existing word. Such software is included in most modern word processors and provides great help in the writing process. For each misspelled word, the spell checker suggests a number of correction suggestions and hopefully, among these is the word intended by the writer. The proposed method evaluates the quality of the correction suggestions given by several popular spell checkers for Swedish.

10.1 Automation and Unsupervision

The evaluation method in this chapter is unsupervised and thus, requires no annotated resources. Instead, it operates on raw, unlabeled text and introduces artificial errors. As a side effect, it makes the evaluation procedure language independent.

10.2 Related Work

The procedure described here is similar to those of Agirre et al. (1998) and Paggio and Underwood (1998) where artificial spelling errors are introduced into text. In (Agirre et al., 1998), the errors are introduced into unrestricted text. In (Paggio and Underwood, 1998), a word-list of correctly spelled words is used for evaluation of lexical coverage while a list of misspelled words is used for evaluation of error coverage. This limits the usability for evaluation of future spell checkers incorporating contextual information.

Agirre et al. (1998) have used Ispell for English in their evaluation while Paggio and Underwood (1998) have used two (anonymous) spell checkers for Danish. We note that Paggio and Underwood (1998) focus on competence errors (e.g. sound-alike errors), while Agirre et al. (1998) focus on performance errors (i.e. keyboard

mistype errors). We have chosen to use performance errors to keep the evaluation procedure language independent.

In the light of the previous work, our contribution is a detailed and thorough investigation of Swedish spell checkers as well as an open-source test bed for unsupervised evaluation of spell checkers, applicable to any language and text type.

10.3 Proposed Method

To evaluate the suggestions given from a spell checker, we will introduce artificial spelling errors into error-free text. To this end, we used the Missplel software to introduce Damerau-type errors (i.e. keyboard mistypes) as described in Section 7.2. One spelling error was introduced per misspelled word. Clearly, to fairly evaluate the spell checkers, an introduced spelling error should only be allowed to result in a non-existing word. Since the original word is known from the error-free text, this is the suggestion we seek from the spell checker.

A spell checker provides correction suggestions in an ordered list. The first position in the list contains the suggestion most likely to be the word intended by the writer. The second suggestion is the second-best fit. The ordering of the suggestions is based on different heuristics, such as word frequency and distance between keyboard keys. Few spell checkers use context to improve the understanding of a misspelled word. However, Agirre et al. (1998) propose several techniques using context to improve spell suggestion (e.g. the use of constraint grammar, Karlsson et al., 1995). Also, the Granska grammar checker does not provide spell correction suggestions leading to a grammatical error.

Given a text containing artificially introduced errors, we applied the spell checker and stored the correction suggestions for all words. We also applied the spell checker on the error-free text. Uppercase and lowercase letters were considered different (so that e.g. 'he' was a different word than 'He'), since most spell checkers support this. Furthermore, if the first word of a sentence is misspelled, you want the suggestions to be capitalized. Also, repeated occurrences of the same misspelled word (common for e.g. proper names) were treated as separate instead of just one occurrence. The reason for this was that not all spell checkers are designed to disregard multiple occurrences.

Given the list of suggestions, we extracted information about how often the original word was the first, second or worse suggestion and how often the word was not suggested at all. Also, we investigated the effect of word length and total suggestion count on the suggestion order. Furthermore, we determined the percentage of false alarms and artificially introduced errors found, the average number of suggestions etc.

10.4 Experiments

The experiments were carried out on the SUC corpus from Section 7.1. We chose to use the same 14 000 words as in the experiments in Chapters 8 and 9. This choice allowed us to determine how well a spell checker could correct the spelling errors automatically on this text (by applying the first suggestion).

In Chapters 8 and 9, the 14 000 words were misspelled with 1%, 2%, 5%, 10% and 20% errors. For each error level, ten files were created containing different artificial spelling errors. In the experiments here, the same 50 files were used as well as the error-free file. Since the spell checkers do not use context, the actual amount of errors in a file is of no relevance. The results from e.g. the 1% level do not differ from say, the 20% level. Hence, the results are presented summarized for all 50 files. However, this is just for the sake of exposition in this chapter; a context-sensitive spell corrector could as easily be evaluated by presenting the error levels separately. The misspelled words found in the error-free file were not included in the calculation for the files containing errors. Since these words were not in the dictionary, they could not be given as a correction suggestion. Instead, they are presented separately to reflect the coverage of the spell checkers' dictionaries.

Three spell checkers were used: Stava (Domeij et al., 1994; Kann et al., 2001), developed at the Department of Numerical Analysis and Computer Science, a free spell checker called Ispell (Kuenning, 1996) and the spell checker in Microsoft Word XP (i.e. Word 2002) (Lingsoft Inc., 2002). Both Stava and Ispell had command-line interfaces while Word was interfaced using a Visual Basic script.

The word lists for Missplel were built from the SUC corpus while none of the spell checkers obtained dictionaries or other information from SUC. Thus, the SUC defined whether or not a word was misspelled. Due to the limited size of SUC, some misspelled words would inevitably be real words. Nevertheless, all programs were faced with this problem and no spell checker benefited or suffered from it more than another.

The results were gathered using AutoEval. The script is provided in Figures 10.4 and 10.5.

10.5 Results

Following the notation of Agirre et al. (1998), the *(error) coverage* of a spell checker is the amount of errors it detects of all errors in the misspelled text. The *precision* is the number of detected errors where the original word is among the suggestions (including errors with no suggestions). The error coverage was 92.2% for Stava, 97.3% for Ispell and 95.5% for Word. The precision was 97.2%, 92.8% and 89.4%, respectively. Following the notation of Paggio and Underwood (1998), the *lexical coverage* is the amount of real words accepted by the spell checker (not marked as errors). This was tested by applying the spell checkers on the supposedly error-free file. The results were 98.2%, 94.8% and 98.3% for Stava, Ispell and Word,

respectively. Hence, the amount of false alarms on error-free text is 100% minus the above lexical coverage.

Regarding the accuracy of the suggestions from the spell checkers, STAVA managed to correctly guess the original word as its first suggestion in 87.6% of the introduced errors. The corresponding numbers for Ispell and Word were 67.4% and 60.0%, respectively. If we consider both the first and second suggestion, the original word was proposed for 95.5% of all misspelled words for STAVA. For Ispell and Word, the numbers were 90.3% and 74.6%, respectively. The ability to correctly guess the original word in the first suggestion was enhanced with increasing word length, as seen in Figure 10.1. Long words have fewer words close to them. However, the data on very long words was sparse and thus, the results fluctuate for words lengths over 20 letters. Furthermore, Ispell never provided a suggestion for words longer than 21 letters and provided very few suggestions for words longer than 15 letters. The reason for this is unknown, but could be due to the fact that Ispell was originally written for English in which long words are rare since there are few compounded words.

Furthermore, the perplexity stemming from more possible suggestions also makes the first suggestion less reliable. For example, having only one suggestion gives an accuracy of 98.8% for STAVA, while having e.g. five suggestion makes the first suggestion correct in 78.1% of the cases. If only one suggestion was given, this was the original word in 95.5% of the cases for Ispell and 97.6% for Word. The amounts of detected errors with exactly one suggestion were 37.9%, 37.0% and 37.6%, respectively. See Figure 10.2 for details on all three spell checkers and the different number of suggestions. The percentage of misspelled words having a particular number of suggestions is reported in Figure 10.3. On the average, the original word was found as suggestion number 1.2 for STAVA, 2.0 for Ispell and 2.3 for Word.

For some of the detected errors, no suggestions were given. For STAVA, this figure was 1.0% of the misspelled words. For Ispell and Word, they were 5.1% and 3.3%, respectively. Furthermore, for some detected errors with suggestions, the original word was not among the suggestions. For STAVA, this figure was 1.8% and for Ispell and Word, the figures were 2.2% and 7.5%, respectively. The average number of suggestions was 3.2, 4.6 and 5.2 for STAVA, Ispell and Word, respectively.

The results are summarized in Table 10.1.

10.6 Discussion

The results from the previous section show different qualities in the spell checkers. Ispell has a very high error coverage (97.3%) but suffers from false alarms (5.2%). Word, on the other hand, has a very low amount of false alarms (1.7%) and a reasonable error coverage (95.5%). STAVA has about the same amount of false alarms (1.8%) and a somewhat lower error coverage (92.2%).

Regarding the suggestions, STAVA is superior at ordering the correction suggestions. The original word is suggested very often, with more than 20 per cent units

	Stava	Ispell	Word
Errors found of all possible (error coverage)	92.2%	97.3%	95.5%
Original word suggested (precision)	97.2%	92.8%	89.4%
Correct words accepted (lexical coverage)	98.2%	94.8%	98.3%
Errors found in error-free text (false alarms)	1.8%	5.2%	1.7%
Word suggested first	87.6%	67.4%	60.0%
Word suggested first or second	95.5%	90.3%	74.6%
Word not suggested	1.8%	2.2%	7.5%
Errors with no suggestions	1.0%	5.1%	3.3%
Single suggestion correct	98.8%	95.5%	97.6%
Errors with a single suggestion	37.9%	37.0%	37.6%
Average suggestion count	3.2	4.6	5.2
Max suggestion count	13	39	20
Average position for correct suggestion	1.2	2.0	2.3

Table 10.1: *Summary of the evaluation of correction suggestions from the spell checkers* STAVA*, Ispell and Word.*

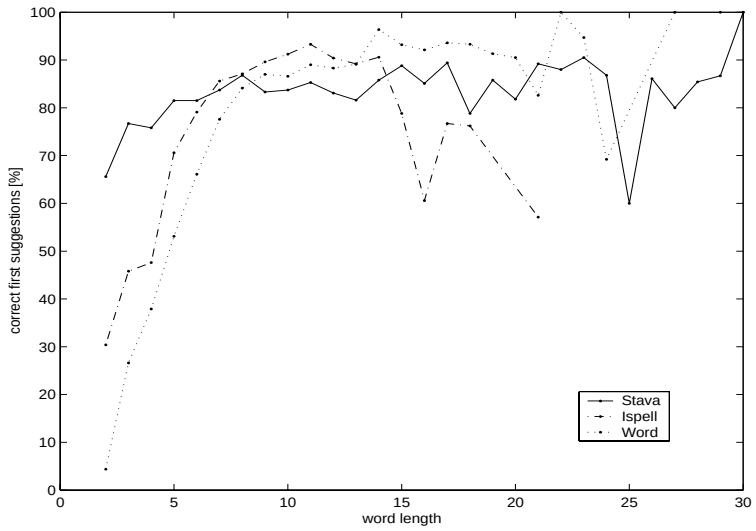

Figure 10.1: *The number of correct first suggestions depending on word length.*

better results for STAVA (87.6%) than the second runner-up Ispell (67.4%). Furthermore, in all other aspects of spelling correction, STAVA has very good performance. It is clearly desirable to have as few suggestions as possible if you can still include the original word, and STAVA has only 3.2 suggestions on the average. We also see that the position in the suggestion list in which the original word appears is much

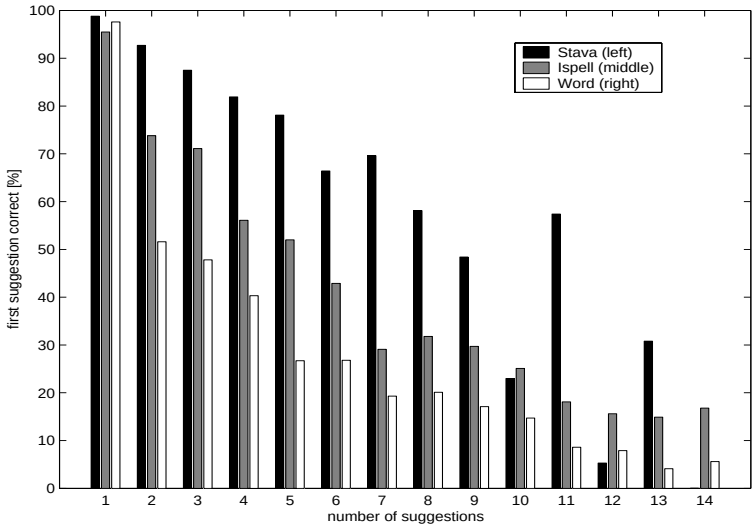

Figure 10.2: *The number of correct first suggestions depending on the suggestion count.*

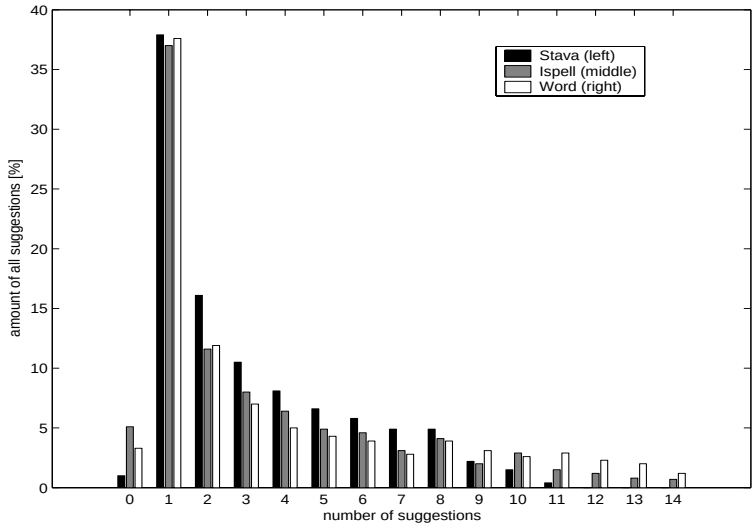

Figure 10.3: *The number of suggestions provided by the spell checkers.*

lower for STAVA than the other programs. In fact, the result 1.2 is very close to one, which would be the perfect result. We conclude that the STAVA algorithm does extremely well on suggesting spelling corrections. We also conclude that the Ispell algorithm has very high error coverage. The algorithm in Word seems to have good error coverage while still avoiding false alarms.

The different levels of error and lexical coverage for the three programs are directly related to dictionary size. A larger dictionary will give lower error coverage but higher lexical coverage. Since a misspelled word is defined by the contents of the SUC corpus, a supposedly misspelled word may be present in the dictionary of one of the spell checkers without being part of the million words of the SUC. Furthermore, differences in error coverage may also be due to different approaches in the analysis of compounded words. As an authentic example in Swedish, 'planering' (*planning*) was misspelled for 'planetring' (*planet ring*), which is an existing word. On the other hand, several examples were encountered where a word such as 'ungdom' (*youth*) was misspelled for 'ugndom', which is not a real word (though it contains the word 'ugn' (*oven*)). The former example should be accepted by a spell checker while the latter should not. Evidently, separating such cases is difficult. Thus, we chose to accept only those words found in the SUC corpus, as mentioned in the Experiments section. Our point is that, to keep the evaluation unsupervised, a spell checker may be penalized despite a correct diagnosis. Nevertheless, a manual check showed that these cases were very rare. Also, in non-compounded languages, such as English, this should not be a problem.

Using Word, we see that for a large amount of words, the original word does not appear as a suggestion (7.5% as compared to about 2% for the other two applications). This is related to choices made in the implementation of the Word spell checker, as the suggestions given from Word always begin with the same letter as the misspelled word. Whether this is a limitation set by the implementation or a design choice based on an assumption that people tend to misspell the first letter of a word more seldom, we do not know. Since the normal use of spell checkers involves spelling errors in the first letter of a word, we chose not to treat this as a special case.

From the results on how often the original word is suggested first, we see that an automatic spell corrector using the STAVA algorithm would have a success rate at 87.6% for words having at least one suggestion. Since 1.2% of the words do not have any suggestion at all, the total success rate would be about $(1 - 0.012) \cdot 0.876 - 0.012 = 85.3\%$. On the other hand, STAVA finds and attempts to correct 1.8% errors in error-free text. Thus, many errors would be eliminated while others would be introduced. We realize that correcting about 85% of the errors would be sufficient to greatly enhance the robustness of the parsers in the evaluations in Chapters 8 and 9. However, this would not measure the robustness of the parsers but rather the correction abilities of the spell checker. Furthermore, the remaining 15% of the errors that are not corrected are actually changed into another word, not intended by the writer. Taggers are normally very robust to spelling errors, much due to accurate statistical heuristics for unknown words. Thus, having a

spell checker correct a word automatically into an unrelated word would make the tagger's work impossible. A word unrelated to the original word could completely throw the tagger and parser off (depending on robustness) and destroy the analysis of a large portion of the context. Experiments in Section 8.6 were carried out to establish the difficulty of parsing corrected text. They showed that applying a parser on auto-corrected spelling errors resulted in lower accuracy than applying the parser on misspelled text. Thus, we have chosen not to correct the spelling errors to keep the error model simple and the evaluation as language independent as possible. Further details on this design choice are given in Section 7.2.

The experiments of Agirre et al. (1998) have many similarities with those carried out here. However, the option of automatic ordering of the spelling suggestions from Ispell was not used, which makes the comparison somewhat difficult. On the other hand, they evaluate a variety of different approaches to correction suggestion and we provide the results of the most successful here. If at least one suggestion was provided, Agirre et al. (1998, pp. 28) observe that when using Ispell for English, the original word is among the suggestions in 100% of the detected errors. This should be compared to the 2.2% of the words where the original word was not suggested for the Swedish Ispell. Even though 100% seems high, the true source of the discrepancy is unknown. Swedish, however, is a compounded language, which may contribute to the difference. In the same paper, we see that the average number of proposals per word is 3.4 for authentic misspellings in text (1257 Ispell proposals for 369 words). On the other hand, the corresponding number of suggestions for artificial errors is 5.6 (7242 + 8083 proposals for 1354 + 1403 words), which is very high compared to 3.4. The value for the Swedish Ispell was 4.6 (and 3.2 for Stava). Since the software AntiSpell used to introduce the error is explained very briefly, we do not know if design choices made there could influence the results. For example, we do not know the weights used to compensate for the fact that human writers tend to confuse keys close to each other on the keyboard more often than those far apart. These weights could affect the authenticity of the introduced errors, but this is only one possible explanation. Other explanations could be a difference in the difficulty or vocabulary of the authentic text and the text used for introduction of artificial spelling errors.

The findings of Agirre et al. (1998) concerning the ability to automatically correct a spelling error for the English Ispell correspond well to the findings for Swedish. It is reported that 80% of the words can receive the correct proposal for English, while for Swedish, Stava can contribute with about 85% correct words. Paggio and Underwood (1998) report that the lexical coverage of the Danish spell checkers was 97% and 99% for the spell checkers denoted A and B, which is comparable to 98.2% for Stava and 98.3% for Word. The Danish spell checkers obtained about 80% and 76% correct suggestions in the first suggestion (calculated from Table 4 in Paggio and Underwood, 1998), provided at least one suggestion. This should be compared to 87.6% for Stava, while it is much higher than 67.4% and 60.0% from Ispell and Word. The Danish spell checkers gave no suggestions for 5.5% and 3.8% of the detected errors, respectively. The Swedish spell checkers had similar

results, except for STAVA with 1.0%. Concerning the error coverage, the Danish spell checkers obtained 99% and 96%, which is good. The corresponding numbers for Swedish were 92.2%, 97.3% and 95.5% for STAVA, Ispell and Word.

We see that most figures for English and Danish are comparable to Swedish and thus, it seems as if the techniques used in modern spelling correction programs are quite language independent in the sense that they are applicable to a variety of western languages. However, it is unfortunate that the order of the spelling corrections from English Ispell was not used in the experiments in Agirre et al. (1998). Ispell was originally written for English and some of the techniques used are supposedly best suited for English. This is further supported by the fact that STAVA, a spell checker originally designed for Swedish, obtains much better results than Ispell (and Word) on suggestion ordering.

This chapter has described an unsupervised evaluation procedure for correction suggestions from spell checkers. From unlabeled text, we can accurately and repeatedly evaluate any aspect of the spelling suggestions. Without manual labor, we have highlighted the strengths and weaknesses of three popular spell checkers for Swedish.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root xmlns="evalcfgfile"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="evalcfgfile eval/cfg.xsd">
  <preprocess>
    infile_plain("zero", "ispell_files/suc.0.ispell");
    infile_plain("in", "ispell_files/suc.10c.ispell");
    infile_plain("err", "files/suc.10c.wte");
    outfile_xml("out", "results/suc.10c.ispell.res");

    NOSUGG = 100;
    IGNORE = 1000;
  </preprocess>
  <process>
    // the error-free file
    field(in("zero"), "\t", "\n", :word0, :sugg0);
    // the misspelled file
    field(in("in"), "\t", "\n", :word1, :sugg1);
    // the file pointing out the introduced errors
    field(in("err"), "\t", "\n", :word2, :tag2, :err2);

    // total number of rows
    ++tot$count;

    // detected error in error-free file
    if(:sugg0 != "ok")
        ++err$in_orig_file;

    // introduced error in misspelled file
    if(:sugg0 == "ok" AND :err2 != "ok")
        ++err$introd_tot;

    // detected error in misspelled file
    if(:sugg0 == "ok" AND :err2 != "ok" AND :sugg1 != "ok")
        ++err$introd_found;

    ...
```

Figure 10.4: AutoEval *configuration for the evaluation of spell checker correction suggestions.*

```
    tmp$sugg_no  = suggestion_number(:word0, :sugg1);
    tmp$sugg_cnt = suggestion_count(:sugg1);
    tmp$word_len = word_length(:word0);
    // ignore word if error found in error-free file
    if(:sugg0 != "ok" OR :err2 == "ok") {
        tmp$sugg_no  = IGNORE;
        tmp$sugg_cnt = IGNORE;
        tmp$word_len = IGNORE;
    }

    // the original word was not among suggestions
    if(tmp$sugg_cnt > 0 AND tmp$sugg_no == NOSUGG)
        ++suggno$word_not_suggested;

    // store position of original word among suggestions
    if(tmp$sugg_no < NOSUGG) {
        ++suggno$("suggno_" . int2str(tmp$sugg_no));
        ++suggno$tot_rows_w_sugg;
    }

    // the origninal word was suggested first
    if(tmp$sugg_no == 1)
        ++suggfirst$("if_cnt_" . int2str(tmp$sugg_cnt));

    // store number of suggestions
    if(tmp$sugg_cnt < NOSUGG)
        ++suggcnt$("cnt_" . int2str(tmp$sugg_cnt));

    // word length n, the original word was sugg first
    if(tmp$sugg_no == 1)
        ++wordlensugg$("first_sugg_if_word_len_" .
                       int2str(tmp$word_len));
    // store word length
    if(tmp$sugg_cnt > 0)
        ++wordlentot$("rows_w_word_len_" .
                      int2str(tmp$word_len));
  </process>
  <postprocess>
    output_all_int(out("out"));
  </postprocess>
</root>
```

Figure 10.5: AutoEval *configuration for the evaluation of spell checker correction suggestions (continued).*

Chapter 11

Semi-supervised Evaluation of ProbCheck

An important objective in the design of an evaluation of a complex system is to minimize the amount of manual work. Due to the many parameters of the ProbCheck algorithm (from Chapter 6), we required a fully automatic evaluation process as close as possible to the situation in which the algorithm is normally used. Clearly, we could produce or use an already existing resource with annotated spelling errors. To produce such a resource would be time-consuming and error-prone. Furthermore, vast amounts of data would be necessary to evaluate the many parameters.

Common spelling errors (e.g. resulting in a non-existing word) are easily detected using a spell checker. Remaining are the context-sensitive spelling errors. As mentioned in the introduction in Section 1.1, many of these are detectable using confusion-set methods. Thus, after applying existing methods, only unpredictable context-sensitive spelling errors resulting from random keyboard mistypes remain. We noted in Section 6.1 that a full parser is a good candidate to detect these errors. The words that do not receive an analysis do not fit into the grammar. Thus, they are probably ungrammatical.

We also noted that sufficient accuracy may be difficult to achieve. For example, in Swedish, both the Uppsala chart parser (Sågvall Hein et al., 2002) and the CLE framework (Gambäck, 1997) have limited coverage. The Malt parser (Nivre et al., 2004) uses a statistical model to assign dependency labels and thus, provides a label for all words. Since no words are left without analysis, Malt is unsuitable for detection of context-sensitive spelling errors.

FDG (Voutilainen, 2001) is a rule-based parser which has reasonable coverage of normal language. The use of rules leaves some of the words without analysis and these words are probably ungrammatical. Hence, we will use FDG as a comparison to the ProbCheck algorithm. As further comparison, we attempt to detect context-sensitive spelling errors using a trigram base-line and a method using tagger transition probabilities (Atwell, 1987). Also, we include a comparison to other

detection methods to establish how a combination of spell and grammar checkers covers the errors made by a human writer.

11.1 Automation and Unsupervision

As stated in Section 1.4, a supervised evaluation involves an annotated resource containing the correct answer for the NLP system output. The evaluation proposed here is semi-supervised since the resource required is not even remotely related to the error detection task of the ProbCheck algorithm. Here, we require a resource annotated with PoS tag information in order to produce context-sensitive spelling errors. The rest of the evaluation procedure is unsupervised.

11.2 Proposed Method

The normal use of the ProbCheck algorithm is to detect context-sensitive spelling errors in text produced by a human writer. Hence, we wanted to simulate this process.

To produce spelling errors closely resembling those of a human writer, we used Missplel from Chapter 4. Missplel was configured to produce keyboard mistype errors resulting in an existing word with a change in PoS tag as discussed in Chapter 7.2. To ascertain that misspelling the word results in a PoS tag change, we require a dictionary with PoS tag information for each word. As an example of context-sensitive spelling error, 'to be or not to be' could be misspelled 'to be or not to me'. This results in a PoS tag change from verb to pronoun and clearly, a context-sensitive spelling error difficult to detect. The results were gathered using AutoEval from Chapter 3.

11.3 Experiments

As described in Chapter 6, ProbCheck uses a parser for phrase transformations. The parser used here was GTA from Chapter 5, a rule-based shallow parser for Swedish. GTA also identified the clause boundaries. The parsing accuracy of GTA is about 88.9% and 88.3% for the clause identification. We used 14 000 words of written Swedish from the SUC corpus from Section 7.1. The text was annotated with parse information, but it was not used here. However, it would be interesting to see how accurate the algorithm is with a perfect parser.

Using Missplel, we introduced errors randomly in 1%, 2%, 5%, 10% and 20% of the words. To minimize the influence of chance, we repeated the process 10 times for each error level, resulting in 50 misspelled texts of 14 000 words each.

Since the algorithm is divided into two parts, PoS tag and phrase transformations, we wanted to assess the individual performance of each part. Thus, each part was turned either on or off, resulting in four different settings. If the PoS transformations were turned off, we simply considered a trigram ungrammatical if

its frequency was below a predetermined threshold e. By turning off both PoS tag and phrase transformations, we obtained a simple trigram base-line, described in Algorithm 1 in Chapter 6.

Furthermore, there were several viable PoS tag similarity measures to use in the statistical error detection. Lee (1999) provides examples of a few, of which we decided to use Jaccard, Jensen-Shannon, L1, L2 and cos:

$$\begin{aligned}
Jac(q,r) &= \frac{|\{v : q(v) > 0 \text{ and } r(v) > 0\}|}{|\{v : q(v) > 0 \text{ or } r(v) > 0\}|} \\
JS(q,r) &= \sum_{v \in V} h(q(v) + r(v)) - h(q(v)) - h(r(v))), \quad h(x) = -x \log x \\
L1(q,r) &= \sum_{v} |q(v) - r(v)| \\
L2(q,r) &= \sqrt{\sum_{v} (q(v) - r(v))^2} \\
cos(q,r) &= \frac{\sum_v q(v) r(v)}{\sum_v q(v)^2 \sum_v r(v)^2} \ ,
\end{aligned}$$

where summation is over all points in the definition set of the probability distributions q and r. For Jensen-Shannon, summation is only over $V = \{v : q(v) > 0 \text{ and } r(v) > 0\}$ to avoid undefined results for the logarithm. For further details on the use of measures, see Section 6.2.

As stated, the arbitrary threshold e was the limit under which trigram frequencies are considered ungrammatical. By setting e to large values, we obtained higher recall from the algorithm and by setting e to small values, we obtained higher precision. In the experiments, we used 12 values of e, namely $e = 0.25, 0.5, 1, 2, 4$ and so forth up to 512.

Clearly, a good understanding of the language simplifies the task of finding context-sensitive spelling errors. A human can easily determine which words are not grammatical. A full parser emulates this knowledge by using a grammar to describe the language. In our case, the parser is FDG (Voutilainen, 2001), a rule-based dependency parser. The words that do not fit into the rules of the grammar are left without analysis by FDG. Thus, these words are probably ungrammatical.

As comparison, we also had a detection algorithm based on tagger transition probabilities. The tagger used was that of Carlberger and Kann (1999), having an accuracy of about 96%. Our comparison method simply used the probabilities provided by the tagger. The tagger determines the most probable PoS tag sequence by using PoS tag trigrams and lexical probabilities. If a word is ambiguous, several PoS tag sequences will be possible. Depending of the weights given by the trigram and lexical probabilities, the most probable PoS tag t will be chosen for a word. The probability of choosing t is defined as the ratio between the weight of t and the weights of the other possible tags stemming from different PoS tag sequences.

The described probabilities seem to offer a decent measure of grammaticality, but present a few problems. For example, a word that has only been observed once with tag r will have probability one despite the sparse data. Furthermore, if two tags are almost equally likely, the chosen tag will receive a probability near or below one half, which is quite low and will indicate ungrammaticality though both PoS tag candidates may be grammatical. Nevertheless, the probability of a PoS tag should give a hint on the grammaticality of a sentence.

The material was scrutinized by the algorithm and the putative errors were marked. Since the minimum resolution of the algorithm is a trigram of words/tags, the algorithm identified the center of the error, and an error within the trigram was deemed correctly identified. The same definition of a detection was also used for FDG and the tagger probabilities. Thus, if a word was detected as an error, it was considered a correct detection if the word on that line was misspelled or a word on an adjacent line was misspelled. From this, we see that the definition of precision and recall will be as follows:

$$\text{recall} = \frac{\#\text{ errors overlapped by any detection}}{\text{total }\#\text{ of introduced errors}},$$

$$\text{precision} = \frac{\#\text{ of detections overlapping an error}}{\text{total }\#\text{ of detection centers}}.$$

11.4 Results

The characteristics of the similarity measures coincided with the findings of Lee (1999), where Jensen-Shannon, L1 and Jaccard were superior to the other measures and had very similar performance. For the sake of exposition, we choose to limit our findings to Jensen-Shannon, which seemed to have a stable performance over all tests. Furthermore, the number of substitution tags m was also a variable. The results showed that $m = 3$ was slightly better than $m = 2$ and $m = 4$ although the results were quite similar. We also saw that $m \geq 5$ resulted in lower performance. Thus, we choose to present only the results for $m = 3$.

The results of the experiments are shown in Figures 11.1 through 11.5 corresponding to the percentage of errors in the text, i.e. 1%, 2%, 5%, 10% and 20%. In each figure, five graphs and one star are displayed. The first four graphs are the combinations of the PoS tags and phrase transformations turned either on or off. The fifth graph is the comparison method. The star represents the FDG parser result. When the error threshold e is increased, the precision drops and the recall increases. We also see that using tagger transition probabilities resulted in poor performance, probably due to the problems mentioned in the previous section.

The ProbCheck algorithm is designed to detect context-sensitive spelling errors. For normal spelling errors and typical grammatical errors, other more suitable algorithms exist. Thus, the proposed algorithm is best used in combination with such algorithms to be able to detect all error types in a text. All algorithms will produce false alarms (i.e. correct text marked as an error), and using more algorithms

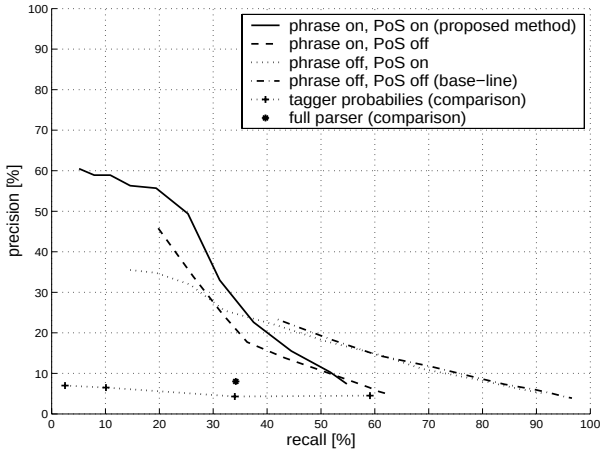

Figure 11.1: *Precision and recall at the 1% error level. The graphs show the four combinations of the PoS tag and phrase transformations turned either on or off, as well as a comparison method using tagger transition probabilities and a comparison method using a full parser (FDG).*

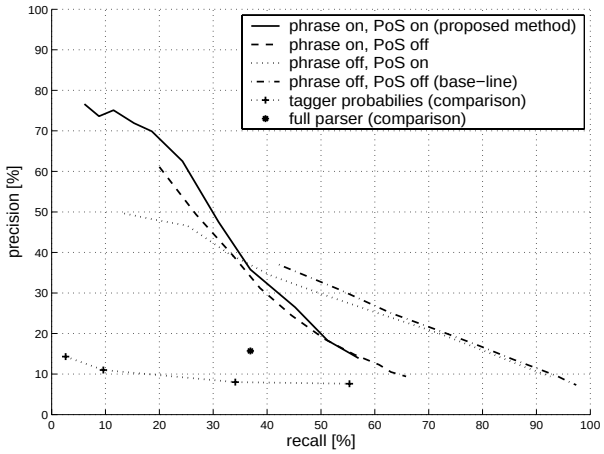

Figure 11.2: *Precision and recall at the 2% error level.*

at the same time will produce more false alarms. Hence, to be able to use the proposed method in combination with others, we want to focus on high precision. The combination of different detection algorithms is described in the next section.

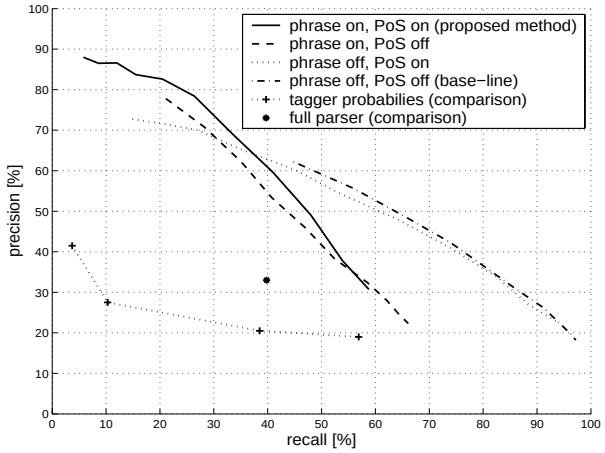

Figure 11.3: *Precision and recall at the 5% error level.*

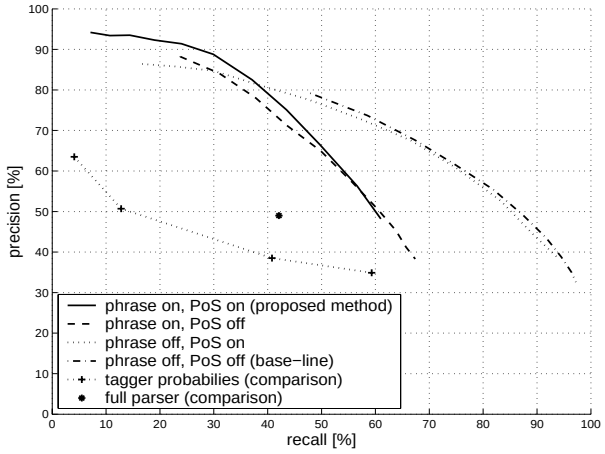

Figure 11.4: *Precision and recall at the 10% error level.*

Normally, only a small amount of the spelling errors in a text are context-sensitive (that is, result in existing words). Peterson (1986) reports that 16% of errors produced by a human may fall into this category (for English, but the results would be similar for Swedish), depending on the size of the dictionary. This is a small fraction of all errors and thus, the 1% and 2% error levels are the most realistic, and the others are shown as comparison.

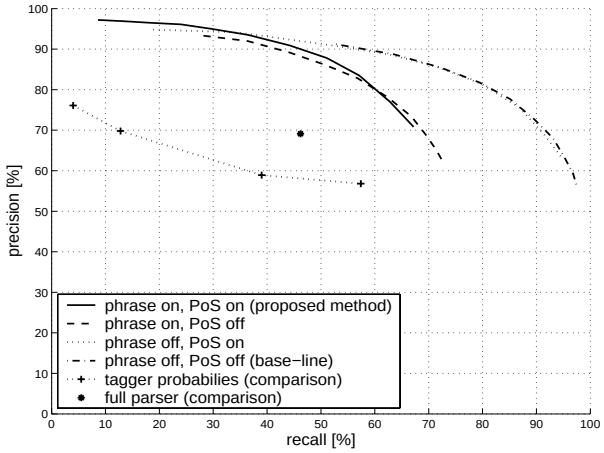

Figure 11.5: *Precision and recall at the 20% error level.*

We see from the figures that using both methods (PoS tag and phrase transformations) obtains the highest precision at all error levels. Furthermore, both PoS tags and phrase transformations contribute to this increase since turning either of them off decreases the precision.

We also see that the base-line (no PoS tags and no phrase transformations) obtains the highest recall, although at a very low precision. When the error levels increase, finding errors is less difficult and the precision increases. This also causes the base-line to obtain a precision closer to the other methods. Nevertheless, at the lower (and realistic) error levels, the proposed method achieves a much higher precision at the expense of a loss in recall. See e.g. Figure 11.1 and compare the highest precision of the base-line (precision 23% at recall 42%) with the proposed method (e.g. precision 50% at recall 26%). Keep in mind here that we cannot expect to achieve high recall while keeping reasonable precision due to the very difficult nature of the errors.

Note that the base-line cannot achieve a lower recall than 23% since this is where $e = 1$. This is the smallest unit in the trigram frequency table.

The results from the comparison methods were always lower than that of the proposed method. The recall of FDG was always near 40%, regardless of error level. This seems to indicate that when randomly introducing Damerau type errors, 40% of the words are very problematic while 60% of the words can be fitted into the grammar. This may be attributed to the fact that the rule-based grammar of FDG must contain rules governing local grammatical constructions, since FDG is relatively robust to many errors (see Chapter 9). Otherwise, an error would destroy the analysis of the whole sentence. On error-free text, FDG found 770 errors amounting to 5.5% false alarms.

	Word	ProbCheck	Granska	ML	All four combined
All detected errors	10	1	8	3	13
All false positives	92	36	35	50	200
Detected spelling errors	8	-	6	1	9
Detected grammatical errors	2	-	2	2	4

Table 11.1: *Evaluation on newspaper texts consisting of 10 000 words.*

	Word	ProbCheck	Granska	ML	All four combined
All detected errors	392	101	411	121	592
All false positives	21	19	13	19	67
Detected spelling errors	334	-	293	26	363
Detected grammatical errors	58	-	118	96	229

Table 11.2: *Evaluation on second language learner essays consisting of 10 000 words.*

11.5 Combining Detection Algorithms

Unrestricted text will inevitably contain a mixture of normal spelling errors, context-sensitive spelling errors as well as grammatical errors. To illustrate the combined use of different error detection techniques, we present the results of a comparative evaluation. The data in this section has been adopted from (Sjöbergh and Knutsson, 2004) with the authors' permission.

Four applications were used in the comparison: the ProbCheck algorithm, the rule-based Swedish grammar checker in Microsoft Word (Arppe, 2000; Birn, 2000), the rule-based Granska grammar checker (Carlberger et al., 2005) and an approach using machine learning (ML) to learn error patterns from artificially introduced errors (Sjöbergh and Knutsson, 2004).

The evaluation was carried out on newspaper text from the Parole corpus (Gellerstam et al., 2000) and text produced by second language learners of Swedish from the SSM corpus (Hammarberg, 1977), each consisting of 10 000 words. The detected errors were checked manually. The rest of the text was not scrutinized by hand.

The results of the evaluation are presented in Tables 11.1 and 11.2. We have chosen not to classify the detected errors from the ProbCheck algorithm since it does not provide a classification of detections.

In Table 11.1, we see that the performance for all algorithms combined on proof-read text was quite low. For example, 13 detected errors and 200 false alarms amounts to a precision of 6.1%. Since not all errors in the text are known, we cannot determine the recall. We also see that the ProbCheck algorithm has few

false alarms in comparison to the other methods.

In Table 11.2 the results on error-prone text are presented. The occurrence of errors has simplified the detection task and thus, the number of false alarms has decreased to from 200 to 67 while precision has increased to 90% (592 detected errors and 67 false alarms).

Evaluation results not presented in the tables show that for the error-prone text, 48 errors are uniquely detected by ProbCheck. Thus, 47% (48 of 101) of the errors detected by ProbCheck contribute to the combined error detector.

11.6 Discussion

Clearly, the performance of the parser and tagger greatly affects the ProbCheck algorithm. Due to the inherent robustness of the PoS tagger, some spelling errors will not result in a change in input to the proposed algorithm. For example, if we introduce 20% errors, only 13.1% of the tags are erroneous from the PoS tagger, as seen in Table 8.1! This results in a lowered recall, since some of the errors are just out of reach for an algorithm working on the output of the PoS tagger. Nevertheless, this is also the situation in normal use of the algorithm.

The ProbCheck algorithm is based on corpus information describing the "language norm". The number of detections from the ProbCheck algorithm is a measure of how well the text conforms to the language norm. Thus, (supposedly) error-free text from different categories may obtain vastly different results. A large number of detections will indicate a complicated text with big discrepancies from the language of the corpus. However, a large number of detections do not necessarily mean ungrammatical language, rather just a complex use of language.

We saw that combining the ProbCheck algorithm with three other applications was successful for text with many errors. The precision for ProbCheck was 90% and about half of the errors detected were not detected by another algorithm. Furthermore, the false alarms from ProbCheck are not necessarily only a nuisance to the user. Detections may signal difficult or complex use of language and the text may benefit from reformulation or rewriting.

Comparing the ProbCheck algorithm to a full parser showed that full language coverage was difficult to achieve. We saw that the precision of ProbCheck was always much higher than the precision for FDG, regardless of error level. We see that even though ProbCheck uses a limited amount of linguistic knowledge and thus, limited amounts of manual work, it achieves good results.

Chapter 12

Concluding Remarks

Most of the thesis work is based upon the two tools AutoEval and Missplel described in the first part of the thesis. While the ideas behind the tools are simple and straightforward, the programs are quite powerful and have been used to successfully implement supervised, semi-supervised and unsupervised evaluation procedures.

The last chapters of the first part of the thesis discussed the development and implementation of two applications: a shallow parser for Swedish called GTA and a detection algorithm for context-sensitive spelling errors called ProbCheck. The shallow parser was based on hand-crafted rules developed in the Granska NLP framework. The parser was also used in the ProbCheck algorithm for phrase transformations. The ProbCheck algorithm used semi-supervised learning to acquire PoS tag distances required for PoS tag transformations. Here, semi-supervision denotes the use of an annotated resource, even though the resource does not explicitly contain the information to be acquired (in this case, the PoS distances).

The second part of the thesis discussed evaluation. The main objective of the work conducted has always been to minimize the amount of manual work. Thus, the most desirable form of evaluation in this respect is unsupervised evaluation. We have focused on three evaluation tasks: evaluating parser robustness, evaluating spell checker correction suggestions and evaluating the ProbCheck algorithm. The end result was an unsupervised evaluation procedure for parser robustness, an unsupervised evaluation for spell checkers and a semi-supervised evaluation procedure for the ProbCheck algorithm.

The results for the ProbCheck algorithm showed that recall had to be sacrificed to gain precision. To cover the full spectrum of spelling and grammatical errors, the algorithm should be used in combination with complementary techniques such as a rule-based grammar checker and a conventional spell checker. Thus, high precision was important since all algorithms introduce false alarms. Considering the very difficult nature of the context-sensitive spelling errors, the performance of the ProbCheck algorithm was acceptable, even though the performance was

somewhat lower than originally expected.

The first attempt to devise a probabilistic error detector in Chapter 6 involved only PoS tag transformations. The recall was good but the precision was low, which to a large extent depended on phrase boundaries producing difficult PoS trigrams. However, the introduction of phrase transformations increased the precision, but reduced the recall more than expected. An alternative approach to PoS tag distances was discussed in Section 6.4. There, it was suggested that the incorporation of left and right context from the scrutinized text could probably increase accuracy of the PoS distances and thus, the performance of the ProbCheck algorithm. Pursuing the ideas of context-sensitive PoS tag distances would probably be rewarding for future work. Another aspect of error detection is the ability to categorize the errors found. Accurately diagnosing an error is important if a detection algorithm is to be considered for commercial use. Many users of modern word processors are second language learners. For these users, the mere ability to detect an error may not be sufficient to correct the error. The ProbCheck algorithm does not offer a categorization for the detected errors although the future implementation of such a categorizer was briefly discussed in Section 6.4. However, the difficult nature of the errors would make a classification difficult.

The unsupervised evaluation for parser robustness (Chapter 9) provided estimates on the degradation of a parser when exposed to noisy input. To assess the quality of the estimates, the results were in turn evaluated using annotated resources. As indicated from the theory behind the unsupervised evaluation, the results were very accurate, with few exceptions. Hence, the proposed method presented a new and accurate means to assess parser robustness without an annotated resource. Using this, different formalisms were compared on the same text. Also, parsers for languages without a large treebank, such as Swedish, could be evaluated for robustness.

We see that the unsupervised evaluation of parser robustness could also perform an automatic analysis of the changes to the parser output due to an artificially introduced error. For example, we could analyze the context of an introduced error to determine how many and how words in the context are affected. Such an analysis would be beneficial to the parser implementer. Clearly, the unsupervised robustness evaluation could also incorporate the detailed analysis of individual phrase types from the supervised robustness evaluation in Chapter 8.

The introduction of artificial spelling errors for parser robustness evaluation was motivated in Section 7.2. However, the spell checker evaluation in Chapter 10 showed that the majority of these errors (85%) could be corrected automatically using a spell checker. The effect of the remaining 15% was also discussed, since correcting a word into an unrelated word could introduce great difficulties such as alternative, but correct parse trees. This is also why we chose not to introduce errors resulting in an existing word. Thus, we used artificial spelling errors without a spelling corrector to keep the error model simple.

An alternative error model could incorporate incomplete sentences, that is, one or more missing words. Clearly, this is a simple and language independent error

model which would be suitable for automatic evaluation. This could be especially suitable to evaluate parsers used in speech applications where restarts and missing words are frequent. However, the parsers used in Chapters 8 and 9 were designed for written language. There, a large amount (say, 5%) of missing words did not seem realistic. Nevertheless, incomplete sentences would be a suitable error model for future work on parsers intended for spoken language.

Developing an error model that is both realistic and language independent would be the ideal solution, but the construction of such a model appears very difficult. MISSPLEL is capable of introducing most errors produced by a human writer. However, determining which error types to introduce is difficult. To determine the amounts of a certain error type, maybe we could resort to a supervised learning algorithm. Given a target domain, such as spoken language, we require a text containing errors annotated with the error type. This could serve as a representation of the error distribution. The task of the supervised learning algorithm is to determine the error types and the relative amounts of errors from each category. The data learnt from the algorithm is the input to MISSPLEL. Using the data, MISSPLEL could be applied on a treebank to produce several texts having the same error distribution as the original, error-prone text, thus reducing the influence of chance. Hence, the text annotated with errors does not need to be annotated with parse information and we do not risk data exhaustion by using the error text repeatedly. Using this, we would obtain language independence and domain specific evaluation. However, the construction of a machine learning algorithm to obtain the error distribution is the real challenge. This would indeed be an interesting topic for future work. Until then, the most realistic and language independent error model available is artificial spelling errors.

To conclude, this thesis has presented several successful automatic methods in NLP. We have presented a novel algorithm for detection of context-sensitive spelling errors. Also, we have provided evaluation procedures producing reliable results while still minimizing or eliminating manual work.

Bibliography

Abney, S., 1991. Parsing by chunks. In R. C. Berwick, S. P. Abney and C. Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics*, pages 257–278. Kluwer Academic Publishers, Boston.

Abney, S., 2002. Bootstrapping. In *Proceedings of ACL 2002*, pages 360–367. Philadelphia, USA.

Agirre, E., K. Gojenola, K. Sarasola and A. Voutilainen, 1998. Towards a single proposal in spelling correction. In *Proceedings of ACL 1998*, pages 22–28. San Francisco, California.

Argamon, A., I. Dagan and Y. Krymolowski, 1998. A memory-based approach to learning shallow natural language patterns. In *Proceedings of the International Conference on Computational Linguistics 1998*, pages 67–73. Association for Computational Linguistics, Montreal, Quebec, Canada.

Arppe, A., 2000. Developing a grammar checker for Swedish. In T. Nordgård, editor, *Proceedings of Nordic Conference in Computational Linguistics 1999*, pages 13–27. Department of Linguistics, University of Trondheim.

Atwell, E., 1987. How to detect grammatical errors in a text without parsing it. In *Proceedings of EACL 1987*, pages 38–45. Copenhagen, Denmark.

Backus, J. W., 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing 1959*, pages 125–132. Paris, France.

Basili, R. and F. M. Zanzotto, 2002. Parsing engineering and empirical robustness. *Natural Language Engineering*, 8(2–3):pages 97–120.

Bigert, J., 2003. The AutoEval and Missplel webpage. http://www.nada.kth.se/theory/humanlang/tools.html.

Bigert, J., 2004. Probabilistic detection of context-sensitive spelling errors. In *Proceedings of LREC 2004*, pages 1633–1636. Lisboa, Portugal.

Bigert, J., 2005. Unsupervised evaluation of spell checker correction suggestions. Forthcoming.

Bigert, J., L. Ericson and A. Solis, 2003a. Missplel and AutoEval: Two generic tools for automatic evaluation. In *Proceedings of the Nordic Conference in Computational Linguistics 2003*. Reykjavik, Iceland.

Bigert, J., V. Kann, O. Knutsson and J. Sjöbergh, 2005a. Grammar checking for Swedish second language learners. In *CALL for the Nordic Languages*, pages 33–47. Samfundslitteratur.

Bigert, J. and O. Knutsson, 2002. Robust error detection: A hybrid approach combining unsupervised error detection and linguistic knowledge. In *Proceedings of Robust Methods in Analysis of Natural Language Data 2002*, pages 10–19. Frascati, Italy.

Bigert, J., O. Knutsson and J. Sjöbergh, 2003b. Automatic evaluation of robustness and degradation in tagging and parsing. In *Proceedings of RANLP 2003*. Bovorets, Bulgaria.

Bigert, J., J. Sjöbergh, O. Knutsson and M. Sahlgren, 2005b. Unsupervised evaluation of parser robustness. In *Proceedings of CICLing 2005*. Mexico City, Mexico.

Birn, J., 1998. Swedish constraint grammar. Technical report, Lingsoft Inc, Helsinki, Finland.

Birn, J., 2000. Detecting grammar errors with lingsoft's Swedish grammar checker. In T. Nordgård, editor, *Proceedings of Nordic Conference in Computational Linguistics 1999*, pages 28–40. Department of Linguistics, University of Trondheim.

Black, E., S. Abney, S. Flickenger, C. Gdaniec, C. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini and T. Strzalkowski, 1991. Procedure for quantitatively comparing the syntactic coverage of English grammars. In *Proceedings of a Workshop on Speech and Natural Language 1991*, pages 306–311. Morgan Kaufmann Publishers Inc., Pacific Grove, California, United States.

Brants, T., 2000. TnT – a statistical part-of-speech tagger. In *Proceedings of ANLP 2000*, pages 224–231. Seattle, USA.

Brill, E., 1992. A simple rule-based part-of-speech tagger. In *Proceedings of ANLP 1992*, pages 152–155. Trento, Italy.

Brodda, B., 1983. An experiment with heuristic parsing of Swedish. In *Proceedings of the EACL 1983*, pages 66–73. Pisa, Italy.

Carlberger, J., R. Domeij, V. Kann and O. Knutsson, 2005. The development and performance of a grammar checker for Swedish: A language engineering perspective. Forthcoming.

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

Carroll, J. and T. Briscoe, 1996. Robust parsing – a brief overview. In *Proceedings of ESSLI 1998*, pages 1–7. Prague, Czech Republic.

Carroll, J., T. Briscoe and A. Sanfilippo, 1998. Parser evaluation: a survey and a new proposal. In *Proceedings of LREC 1998*, pages 447–454. Granada, Spain.

Chomsky, N., 1956. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):pages 113–124.

Clark, A., 2001. *Unsupervised Language Acquisition: Theory and Practice*. Ph.D. thesis, COGS, University of Sussex.

Collins, M., J. Hajic, L. Ramshaw and C. Tillmann, 1999. A statistical parser for Czech. In *Proceedings of the Annual Meeting of the ACL 1999*. College Park, Maryland.

Daelemans, W., J. Zavrel, K. van der Sloot and A. van den Bosch, 2001. TiMBL: Tilburg memory-based learner – version 4.0 reference guide. http://ilk.kub.nl/software.html.

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

Domeij, R., J. Hollman and V. Kann, 1994. Detection of spelling errors in Swedish not using a word list en clair. *Journal of Quantitative Linguistics*, 1(3):pages 195–201.

Domeij, R., O. Knutsson, J. Carlberger and V. Kann, 2000. Granska – an efficient hybrid system for Swedish grammar checking. In T. Nordgård, editor, *Proceedings of Nordic Conference in Computational Linguistics 1999*, pages 28–40. Department of Linguistics, University of Trondheim.

Ejerhed, E., 1999. Finite state segmentation of discourse into clauses. In A. Kornai, editor, *Extended Finite State Models of Language*, chapter 13. Cambridge University Press.

Ejerhed, E., G. Källgren, O. Wennstedt and M. Åström, 1992. *The Linguistic Annotation System of the Stockholm-Umeå Project*. Department of Linguistics, University of Umeå, Sweden.

Ericson, L., 2004. Missplel – a generic tool for introduction of spelling errors. Master's thesis (in Swedish), Royal Institute of Technology, Stockholm, Sweden, TRITA-NA-E04045.

Foster, J., 2004. Parsing ungrammatical input: An evaluation procedure. In *Proceedings of LREC 2004*, pages 2039–2042. Lisbon, Portugal.

Gale, W. and K. Church, 1993. A program for aligning sentences in bilingual corpora. *Computational Linguistics*, 19(1):pages 75–102.

Gambäck, B., 1997. *Processing Swedish Sentences: A Unification-Based Grammar and some Applications*. Ph.D. thesis, The Royal Institute of Technology and Stockholm University.

Gellerstam, M., Y. Cederholm and T. Rasmark, 2000. The bank of Swedish. In *Proceedings of LREC 2000*, pages 329–333. Athens, Greece.

Golding, A., 1995. A Bayesian hybrid method for context-sensitive spelling correction. In D. Yarovsky and K. Church, editors, *Proceedings of the Workshop on Very Large Corpora 1995*, pages 39–53. Somerset, New Jersey.

Golding, A. and D. Roth, 1996. Applying winnow to context-sensitive spelling correction. In *Proceedings of the International Conference on Machine Learning 1996*, pages 182–190. Bari, Italy.

Golding, A. and D. Roth, 1999. A winnow-based approach to context-sensitive spelling correction. *Machine Learning*, 34(1-3):pages 107–130.

Golding, A. and Y. Schabes, 1996. Combining trigram-based and feature-based methods for context-sensitive spelling correction. In A. Joshi and M. Palmer, editors, *Proceedings of ACL 1996*, pages 71–78. San Francisco, USA.

Grishman, R., C. Macleod and J. Sterling, 1992. Evaluating parsing strategies using standardized parse files. In *Proceedings of ANLP 1992*, pages 156–161. Trento, Italy.

Grudin, J., 1981. *The organization of serial order in typing*. Ph.D. thesis, Univ. of California, San Diego.

Hammarberg, B., 1977. Svenskan i ljuset av invandrares språkfel (Swedish in the light of errors made by second language learners). In *Nysvenska studier 57*, pages 60–73.

Hammerton, J., M. Osborne, S. Armstrong and W. Daelemans, 2002. Introduction to special issue on machine learning approaches to shallow parsing. *Journal of Machine Learning Research*, Special Issue on Shallow Parsing(2):pages 551–558.

Hogenhout, W. I. and Y. Matsumoto, 1996. Towards a more careful evaluation of broad coverage parsing systems. In *Proceedings of COLING 1996*, pages 562–567. San Francisco, USA.

Järvinen, T. and P. Tapanainen, 1997. A dependency parser for English. Technical report, Department of Linguistics, University of Helsinki.

Jones, M. and J. Martin, 1997. Contextual spelling correction using latent semantic analysis. In *Proceedings of the ANLP 1997*, pages 166–173. Washington, DC.

Källgren, G., 1991. Parsing without lexicon: the MorP system. In *Proceedings of the EACL 1991*, pages 143–148. Berlin, Germany.

Kann, V., R. Domeij, J. Hollman and M. Tillenius, 2001. Implementation aspects and applications of a spelling correction algorithm. In *Text as a Linguistic Paradigm: Levels, Constituents, Constructs*, volume 60 of *Quantitative Linguistics*, pages 108–123.

Karlsson, F., A. Voutilainen, J. Heikkilä and A. Anttila, 1995. *Constraint Grammar. A Language Independent System for Parsing Unrestricted text*. Mouton de Gruyter, Berlin, Germany.

King, M. et al., 1995. EAGLES – evaluation of natural language processing systems. http://issco-www.unige.ch/ewg95.

Knutsson, O., J. Bigert and V. Kann, 2003. A robust shallow parser for Swedish. In *Proceedings of the Nordic Conference in Computational Linguistics 2003*. Reykjavik, Iceland.

Knutsson, O., T. Cerratto Pargman, K. Severinson Eklundh and S. Westlund, 2004. Designing and developing a language environment for second language writers. *Forthcoming*.

Kokkinakis, D. and S. Johansson-Kokkinakis, 1999. A cascaded finite-state parser for syntactic analysis of Swedish. In *Proceedings of EACL 1999*, pages 245–248. Bergen, Norway.

Kuenning, G., 1996. International Ispell, Swedish dictionaries by Göran Andersson and SSLUG. http://fmg-www.cs.ucla.edu/fmg-members/geoff/ispell.html.

Lang, B., 1988. Parsing incomplete sentences. *Proceedings of COLING 1988*, pages 365–371.

Lee, L., 1999. Measures of distributional similarity. In *Proceedings of ACL 1999*, pages 25–32.

Li, X. and D. Roth, 2001. Exploring evidence for shallow parsing. In W. Daelemans and R. Zajac, editors, *Proceedings of CoNLL 2001*, pages 38–44. Toulouse, France.

Lin, D., 1995. A dependency-based method for evaluating broad-coverage parsers. In *Proceedings of IJCAI 1995*, pages 1420–1427. Montreal, Quebec, Canada.

Lin, D., 1998. A dependency-based method for evaluating broad-coverage parsers. *Natural Language Engineering*, 4(2):pages 97–114.

Lingsoft Inc., 2002. From the help in Microsoft Word: "Swedish grammar checker, spell checker, syllabification and inflecting thesaurus by Lingsoft inc.".

Maegaard, B. et al., 1997. TEMAA – a testbed study of evaluation methodologies: Authoring aids. http://cst.dk/projects/temaa/temaa.html.

Mayberry, M., 2004. *Incremental Nonmonotonic Parsing through Semantic Self-Organization*. Ph.D. thesis, University of Austin, Texas.

Megyesi, B., 2002a. *Data-Driven Syntactic Analysis – Methods and Applications for Swedish*. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden.

Megyesi, B., 2002b. Shallow parsing with PoS taggers and linguistic features. *Journal of Machine Learning Research*, Special Issue on Shallow Parsing(2):pages 639–668.

Menzel, W., 1995. Robust processing of natural language. In *Proceedings of the Annual German Conference on Artificial Intelligence 1995*, pages 19–34. Berlin, Germany.

Miikkulainen, R., 1996. Subsymbolic case-role analysis of sentences with embedded clauses. *Cognitive Science*, 20(1):pages 47–73.

Munoz, M., V. Punyakanok, D. Roth and D. Zimak, 1999. A learning approach to shallow parsing. In *In Proceedings of EMNLP-VLC 1999*, pages 168–178. Maryland, USA.

Netter, K. et al., 1998. DiET – diagnostic and evaluation tools for natural language applications. In *Proceedings of LREC 1998*, pages 573–579. Grenada, Spain.

Ngai, G. and R. Florian, 2001. Transformation-based learning in the fast lane. In *Proceedings of NAACL 2001*, pages 40–47. Carnegie Mellon University, Pittsburgh, USA.

Nivre, J., 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of IWPT 2003*, pages 149–160. Nancy, France.

Nivre, J., J. Hall and J. Nilsson, 2004. Memory-based dependency parsing. In *Proceedings of CoNLL 2004*, pages 49–56. Boston, USA.

Paggio, P. and N. Underwood, 1998. Validating the TEMAA LE evaluation methodology: a case study on Danish spelling checkers. *Natural Language Engineering*, 4(3):pages 211–228.

Pearce, D. and H.-G. Hirsch, 2000. The Aurora experimental framework for the performance evaluation of speech recognition systems under noisy conditions. In *Proceedings of International Conference on Spoken Language Processing 2000*, pages 29–32. Beijing, China.

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

Radford, A., 1988. *Transformational Grammar*. Cambridge University Press, Cambridge.

Rajman, M. et al., 1999. ELSE – evaluation in language and speech engineering. http://www.limsi.fr/TLP/ELSE/.

Ramshaw, L. and M. Marcus, 1995. Text chunking using transformation-based learning. In D. Yarovsky and K. Church, editors, *Proceedings of Workshop on Very Large Corpora 1995*, pages 82–94. Somerset, New Jersey.

Ratnaparkhi, A., 1996. A maximum entropy part-of-speech tagger. In *Proceedings of EMNLP 1996*, pages 133–142. Somerset, New Jersey.

Saito, H. and M. Tomita, 1988. Parsing noisy sentences. *Proceedings of COLING 1988*, pages 561–566.

Schmid, H., 1994. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of International Conference on New Methods in Language Processing 1994*, pages 172–176. Manchester, UK.

Sjöbergh, J. and O. Knutsson, 2004. Faking errors to avoid making errors: Very weakly supervised learning for error detection in writing. Forthcoming.

Solis, A., 2003. AutoEval – a generic tool for automatic evaluation of natural language applications (in Swedish). Master's thesis, Royal Institute of Technology, Stockholm, Sweden, TRITA-NA-E03012.

Sågvall Hein, A., 1982. An experimental parser. In *Proceedings of COLING 1982*, pages 121–126. Prague, Czech Republic.

Sågvall Hein, A., A. Almqvist, E. Forsbom, J. Tiedemann, P. Weijnitz, L. Olsson and S. Thaning, 2002. Scaling up an MT prototype for industrial use. Databases and data flow. In *Proceedings of LREC 2002*, pages 1759–1766. Las Palmas, Spain.

Srinivas, B., C. Doran, B. Hockey and A. Joshi, 1996. An approach to robust partial parsing and evaluation metrics. In *Proceedings of ESSLI 1996*. Prague, Czech Republic.

Star Trek Voyager, 1995–2001. TV series. The computer voice of starship USS Voyager is that of actress Majel Barrett, who played nurse Christine Chapel in the original Star Trek series.

Tesnière, L., 1959. Éléments de syntaxe structurale. In *Libraire C*. Klincksieck, Paris.

Tjong Kim Sang, E. and S. Buchholz, 2000. Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of CoNLL 2000 and LLL 2000*, pages 127–132. Lisbon, Portugal.

Tjong Kim Sang, E. F., 2000. Noun phrase representation by system combination. In *Proceedings of ANLP-NAACL 2000*, pages 335–336. Seattle, Washington, USA.

Vilares, M., V. Darriba and J. Vilares, 2004. Parsing incomplete sentences revisited. In *Proceedings of CICLing 2004*, pages 102–111. Seoul, Korea.

Vilares, M., V. M. Darriba, J. Vilares and R. Rodriguez, 2003. Robust parsing using dynamic programming. In *Proceedings of the Conference on Implementation and Application of Automata (CIAA) 2003*, pages 258–267. Santa Barbara, CA, USA.

Voutilainen, A., 2001. Parsing Swedish. In *Proceedings of the Nordic Conference in Computational Linguistics 2001*. Uppsala, Sweden.

Yarowsky, D., 1994. Decision lists for lexical ambiguity resolution: Application to accent restoration in Spanish and French. In *Proceedings of ACL 1994*, pages 88–95. Las Cruces, New Mexico, USA.