Växjö University

Johan Hall

MaltParser – An Architecture for Inductive Labeled Dependency Parsing

Licentiate Thesis

School of Mathematics and Systems Engineering

Reports from MSI

MaltParser – An Architecture for Inductive Labeled Dependency
Parsing

Johan Hall

MaltParser – An Architecture for Inductive Labeled Dependency Parsing

Licentiate Thesis

Computer Science

2006

Växjö University

A thesis for the Degree of Licentiate of Philosophy in Computer Science at Växjö University.

MaltParser – An Architecture for Inductive Labeled Dependency Parsing
Johan Hall

Växjö University

School of Mathematics and Systems Engineering

SE-351 95 Växjö, Sweden

http://www.vxu.se/msi

To Gert and Karin

Abstract

This licentiate thesis presents a software architecture for inductive labeled dependency parsing of unrestricted natural language text, which achieves a strict modularization of parsing algorithm, feature model and learning method such that these parameters can be varied independently. The architecture is based on the theoretical framework of inductive dependency parsing by Nivre (2006) and has been realized in MaltParser, a system that supports several parsing algorithms and learning methods, for which complex feature models can be defined in a special description language. Special attention is given in this thesis to learning methods based on support vector machines (SVM).

The implementation is validated in three sets of experiments using data from three languages (Chinese, English and Swedish). First, we check if the implementation realizes the underlying architecture. The experiments show that the MaltParser system outperforms the baseline and satisfies the basic constraints of well-formedness. Furthermore, the experiments show that it is possible to vary parsing algorithm, feature model and learning method independently. Secondly, we focus on the special properties of the SVM interface. It is possible to reduce the learning and parsing time without sacrificing accuracy by dividing the training data into smaller sets, according to the part-of-speech of the next token in the current parser configuration. Thirdly, the last set of experiments present a broad empirical study that compares SVM to memory-based learning (MBL) with five different feature models, where all combinations have gone through parameter optimization for both learning methods. The study shows that SVM outperforms MBL for more complex and lexicalized feature models with respect to parsing accuracy. There are also indications that SVM, with a splitting strategy, can achieve faster parsing than MBL. The parsing accuracy achieved is the highest reported for the Swedish data set and very close to the state of the art for Chinese and English.

Key-words: Dependency Parsing, Support Vector Machines, Machine Learning.

Sammanfattning

Denna licentiatavhandling presenterar en mjukvaruarkitektur för datadriven dependensparsning, dvs. för att automatiskt skapa en syntaktisk analys i form av dependensgrafer för meningar i texter på naturligt språk. Arkitekturen bygger på idén att man ska kunna variera parsningsalgoritm, särdragsmodell och inlärningsmetod oberoende av varandra. Till grund för denna arkitektur har vi använt det teoretiska ramverket för induktiv dependensparsning presenterat av Nivre (2006). Arkitekturen har realiserats i programvaran MaltParser, där det är möjligt att definiera komplexa särdragsmodeller i ett speciellt beskrivningsspråk. I denna avhandling kommer vi att lägga extra tyngd vid att beskriva hur vi har integrerat inlärningsmetoden supportvektormaskiner (SVM).

MaltParser valideras med tre experimentserier, där data från tre språk används (kinesiska, engelska och svenska). I den första experimentserien kontrolleras om implementationen realiserar den underliggande arkitekturen. Experimenten visar att MaltParser utklassar en trivial metod för dependensparsning (*eng.* baseline) och de grundläggande kraven på välformade dependensgrafer uppfylls. Dessutom visar experimenten att det är möjligt att variera parsningsalgoritm, särdragsmodell och inlärningsmetod oberoende av varandra. Den andra experimentserien fokuserar på de speciella egenskaperna för SVM-gränssnittet. Experimenten visar att det är möjligt att reducera inlärnings- och parsningstiden utan att förlora i parsningskorrekthet genom att dela upp träningsdata enligt ordklasstaggen för nästa ord i nuvarande parsningskonfiguration. Den tredje och sista experimentserien presenterar en empirisk undersökning som jämför SVM med minnesbaserad inlärning (MBL). Studien använder sig av fem särdragsmodeller, där alla kombinationer av språk, inlärningsmetod och särdragsmodell har genomgått omfattande parameteroptimering. Experimenten visar att SVM överträffar MBL för mer komplexa och lexikaliserade särdragsmodeller med avseende på parsningskorrekthet. Det finns även vissa indikationer på att SVM, med en uppdelningsstrategi, kan parsa en text snabbare än MBL. För svenska kan vi rapportera den högsta parsningskorrektheten hittills och för kinesiska och engelska är resultaten nära de bästa som har rapporterats.

Acknowledgments

Unfortunately, my parents Gert and Karin cannot witness the completion of my licentiate thesis, but I know that they would be very proud of me. They always believed in me and supported me in whatever I wanted to do. I want to thank my supervisor Joakim Nivre for all fruitful discussions, advice and fun times when we developed MaltParser and I am looking forward to the development of next version. I especially want to thank Jens Nilsson for the conversion of all the data used in this thesis into dependency structures and for the MaltEval tool, which made it easier to validate the MaltParser system. For the conversion of the Chinese data we used the head rules made by Yuan Ding at the University of Pennsylvania. I also want to thank all my colleagues in computer science at Växjö University to make it fun to go to work every day. I especially want to thank Morgan Ericsson for many ideas and extra computer power.

Finally, I want to thank my love Kristina for all support when I wrote this thesis.

ix

Contents

Contents

xii

Chapter 1

Introduction

Syntactic parsing is an important component for many applications of natural language processing. In this thesis, we regard *parsing* as the process of mapping sentences in unrestricted natural language text to their syntactic representations. Furthermore, the program which performs this process is called a *syntactic parser*, or simply *parser*. The syntactic structure is formalized with a syntactic representation such as *phrase structure* or *dependency structure*. Parsing a sentence with phrase structure grammar or context-free grammar recursively decomposes it into constituents or phrases and in that way a phrase structure tree is created with relationships between words and phrases. By contrast, with dependency structure representations, the goal of parsing a sentence is to create a dependency graph consisting of lexical nodes linked by binary relations called *dependencies*. A dependency relation connects words with one word acting as *head* and the other as *dependent*. In this thesis, we will concentrate on parsing with dependency representations.

Data-driven methods in natural language processing have been used in many tasks in the past decade and syntactic parsing is one of them. Statistical parsing is usually based on nondeterministic parsing techniques in combination with generative probabilistic models that provide an n-best ranking of the set of candidate analyses derived by the parser (Collins 1997; Collins 1999; Charniak 2000). Discriminative models can be used to enhance these parsers by reranking the analyses output by the parser (Johnson et al. 1999; Collins and Duffy 2005; Charniak and Johnson 2005).

Nondeterministic parsing has been the mainstream approach, but it has also been shown that deterministic parsing can be performed with fairly high accuracy, especially in dependency-based parsing (Kudo and Matsumoto 2000a; Yamada and Matsumoto 2003; Nivre et al. 2004; Isozaki et al. 2004; Cheng et al. 2005a), but also in constituent-based parsing (Sagae and Lavie 2005). The main idea is to guide the parser with a classifier trained on treebank data using a greedy parsing algorithm that approximates a globally optimal solution by making a series of locally optimal choices. A deterministic parser usually uses a form of history-based feature model (Black et al.

1992; Magerman 1995; Ratnaparkhi 1997) to create a representation that a classifier can use to predict the next parser state. This is also the approach assumed in this thesis.

Availability of large syntactically annotated corpora, also known as *tree-banks*, is essential when constructing data-driven parsers, but one of the potential advantages is that they can easily be ported to new languages. A problem is that many data-driven parsers are overfitted to a particular language, usually English. For example, Corazza et al. (2004) report increased error rates of 15–18% when using two statistical parsers developed for English to parse Italian. We suggest that a data-driven parser need to be designed for flexible reconfiguration to increase the portability to other languages. A user should be able to experiment with several parsing algorithms, feature models and learning methods.

1.1   Research Problem and Aims

The main research problem for the doctoral thesis is to study the influence of different factors on accuracy and efficiency of data-driven dependency parsing. This study requires a broad evaluation of the parsing system, where we perform extensive feature selection and parameter tuning to optimize the feature models and classifiers for many languages and several parsing algorithms.

For the licentiate thesis we will restrict the research problem to the design, implementation and validation of an architecture for data-driven dependency parsing of unrestricted natural language text. The validation can be seen as a pilot evaluation that will determine future directions. However, we will also obtain experimental results that have a direct bearing on the long-term research problems.

We present a software architecture that should be able to handle different parsing algorithms, feature models and learning methods, for both learning and parsing. When using the implementation of this architecture, the user should be able to vary these parameters independently in a convenient way.

The choice of parsing algorithm influences how the syntactic structure will be built. In the learning phase this will affect how the training data is generated and in the parsing phase which structures are permissible. It should be easy to add new parsing algorithms into the architecture, provided that they fulfil certain well-defined requirements.

The linguistic knowledge of the language is important when defining the structure of the feature model, in other words which linguistic features should be used to predict parsing actions. It should be easy to define a new feature

2

model without reprogramming the system. A feature model should be defined in an appropriate feature specification language so that it can be loaded when it is required.

Given a set of training instances, where each instance is a fingerprint of the current state of the parser, as specified by the feature model, together with the transition to the next parser state, the task of the learner is to induce a model at learning time. At parsing time, this model is then used for predicting the next parser state. This task can easily be formulated as a classification task, where discriminative learning methods are well-suited.

The architecture is based on the theoretical framework of inductive dependency parsing by Nivre (2006) and has been realized in a system called Malt-Parser (Nivre et al. 2006), which in the current version supports two parsing algorithms, in several versions, and two learning methods (MBL and SVM), for which complex feature models can be defined in a special description language. The implementation of the MaltParser system has been joint work together with Joakim Nivre. MaltParser was first equipped with an interface to a memory-based learner called TiMBL (Daelemans and Van den Bosch 2005) and Nivre (2006) contains an extensive evaluation of memory-based dependency parsing using the parsing algorithm defined in Nivre (2003). In order to validate the generality and flexibility of the architecture, we therefore have to extend the parser with an interface to another learner and implement an additional deterministic parsing algorithm. We have chosen to use SVM as the learning method, because it has been proven to give good results for similar tasks (Kudo and Matsumoto 2000b; Yamada and Matsumoto 2003; Sagae and Lavie 2005). For the parsing algorithm, we have chosen the incremental algorithm described in Covington (2001).

Using this new implementation, we have performed three sets of experiments, designed to answer three essential questions:

1. **Validation of the implementation**: Does MaltParser realize the underlying architecture, so that it is possible to vary parsing algorithm, feature model and learning method independently?

2. **Investigation of the SVM interface**: How do the special properties of the SVM interface affect parsing accuracy and time efficiency? How can learning and parsing efficiency be improved without sacrificing accuracy?

3. **Comparison of MBL and SVM**: Which of the learning methods is best suited for the task of inductive labeled dependency parsing, taking both parsing accuracy and time efficiency into account?

3

Apart from answering these questions, we will try to identify future directions that hopefully will be useful for the long-term research problems.

1.2 Outline of the Thesis

In this introductory chapter, we have tried to outline the long-term research problem and the specific aims of the licentiate thesis. The structure of the remaining chapters is as follows.

Chapter 2, Inductive Dependency Parsing
Chapter 2 reviews the background material for this thesis. We define the problem of parsing unrestricted natural language text and discuss different algorithms for dependency parsing. Furthermore, data-driven parsing and especially the history-based models are discussed. The chapter continues with a description of the two machine learning methods used in the rest of the thesis: SVM and MBL. Finally, the chapter ends with a section which briefly presents related work.

Chapter 3, MaltParser
Chapter 3 presents an architecture for parsing unrestricted natural language text with dependency structures. The architecture is described in detail with focus on the two main modules *Parser* and *Guide*. The MaltParser system is an implementation of the architecture and the chapter ends with a description of this system.

Chapter 4, Experiments
Chapter 4 starts with a presentation of the treebank data used for the experiments and an explanation of the evaluation criteria used to validate the implementation of the proposed architecture. An investigation of the three questions explained above is presented based on extensive experiments.

Chapter 5, Conclusion
Chapter 5 contains the main conclusions and a summary of the main results of the thesis. The chapter ends with a discussion of directions for future research.

1.3 Division of Labor

As already stated, the design and implementation of MaltParser is joint work with Joakim Nivre. More specifically, the work has been divided as follows:

4

- The design of the architecture is joint work.

- The implementation of parsing algorithms, generic feature model handling and the memory-based learner is mainly the work of Joakim Nivre.

- The implementation of all other parts of the system, including the SVM learner, is mainly the work of Johan Hall.

5

Chapter 2

Background

Syntactic parsing is used in many applications such as machine translation, information extraction and question answering. Applications dealing with unrestricted text need to handle all kinds of text, including grammatically correct text, ungrammatical text and foreign expressions. It is desirable that such an application produces some kind of analysis. Of course, if the input is "garbage", it is most likely that the system will fail to create an interesting analysis, but the system should nevertheless make its best to produce an analysis. If these applications need a syntactic parser, it also needs to be able to handle unrestricted text, although we need to restrict the text to a certain natural language to be able to derive a meaningful syntactic representation. Nivre (2006) introduces the notion of *text parsing* to characterize this open-ended problem that can only be evaluated with respect to empirical samples of a *text language*.[1]

Our approach to text parsing is dependency-based and data-driven. The goal of dependency-based text parsing is to construct a dependency graph for each sentence in a text. Figure 2.1 shows an example of a dependency graph, connecting the words in a Swedish sentence by binary relations labeled with dependency types (grammatical functions).

Data-driven methods comply well with the fact that text parsing uses empirical samples of a *text language*. A realistic approach is then to use some kind of supervised learning method that makes use of a treebank, which consists of syntactically annotated sentences. A problem with this approach is that it restricts us to languages that have at least one treebank. In addition, these treebanks are often annotated with constituency-based representations and therefore need to be converted to dependency-based representations.

Given that we have a treebank for a specific language our approach is to induce a parser model at learning time and use this parser model to parse sentences. However, since it is problematic to use the dependency graph directly

---

[1]The term *text language* does not exclude spoken language, but emphasizes that it is a language that occurrs in real texts. In principle, the notion applies also to utterances in spoken dialogue.
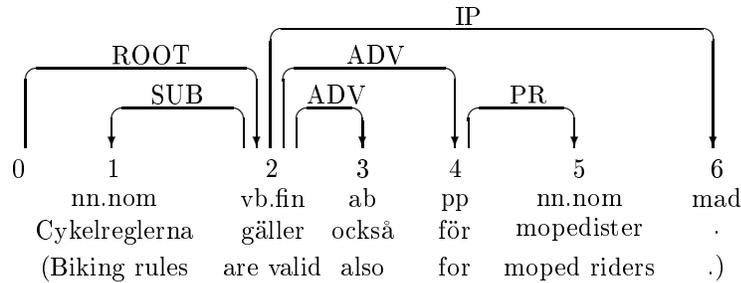
Figure 2.1: Dependency graph for Swedish sentence, converted from Tal-banken

to construct such a model, we instead use a deterministic parsing algorithm to map a dependency graph to a transition sequence such that this transition sequence uniquely determines the dependency graph. An individual transition can be, for example, shifting a token onto a stack or adding an arc between two tokens. The transition system in itself is normally nondeterministic and we therefore need a mechanism that resolves this nondeterminism. We use a discriminative learning method, such as SVM and MBL, to construct a classifier. Moreover, we use history-based feature models to extract vectors of feature-value pairs from the current parser state as training material for the classifier.

In this chapter, we review the necessary background for the design and implementation of MaltParser focusing on the framework of inductive dependency parsing proposed by Nivre (2006). Most of the notation used by Nivre (2006) is also used here, but in some cases the notation has to be extended. The rest of the chapter is structured as follows. Section 2.1 describes the basic requirements on text parsing. Section 2.2 presents the necessary definitions of dependency graphs. Section 2.3 presents the parsing framework, including the deterministic parsing algorithm, the history-based feature models and discriminative learning methods. Related work is discussed in section 2.4.

2.1 Requirements on Text Parsing

We begin by defining a text as a sequence $T = (x_1, \ldots, x_n)$ of sentences, where each sentence $x_i = (w_1, \ldots, w_m)$ is a sequence of tokens and a token

8

w_j is a sequence of characters, usually a word form. Given a text T, the task of text parsing is to derive the correct analysis y_i for every sentence $x_i \in T$. We assume that the text T contains sentences of a text language L that in our case is a natural language. This assumption entails that the text language is not a formal language and that parsing does not entail recognition. Instead, we see text parsing as an empirical approximation problem. Therefore, this approach is not well-suited for grammar checking in a word-processing application, because it will try to find an analysis also for an ungrammatical sentence.

Given these definitions we can define four basic requirements on a text parser (Nivre 2006):

Definition 2.1. A parser P should map a text $T = (x_1, \ldots, x_n)$ in language L to well-formed syntactic representations $(y_1, \ldots, y_n)$ in a way that satisfies the following requirements:

1. **Robustness**: P assigns at least one analysis y_i to every sentence $x_i \in T$.

2. **Disambiguation**: P assigns at most one analysis y_i to every sentence $x_i \in T$.

3. **Accuracy**: P assigns the correct analysis y_i to every sentence $x_i \in T$.

4. **Efficiency**: P processes every sentence $x_i \in T$ in time and space that is polynomial in the length of x_i.

We want to create a parser that uses a parsing strategy that assigns at least one analysis for each sentence (**Robustness**) and at most one analysis (**Disambiguation**). The third requirement (**Accuracy**) is unrealistic in practice, but we will use this as an evaluation criterion in the Chapter 4. In order to satisfy the fourth requirement, we will use deterministic parsing algorithms with at most quadratic time complexity and linear space complexity. We will use two parsing algorithms that have linear complexity (Nivre's arc-eager and arc-standard algorithms) and one that has quadratic complexity (Covington's algorithm). In the experiments, the **Efficiency** requirement will be an evaluation criterion that measures the time it takes to parse a text.

2.2 Dependency Graphs

Dependency parsing is based on syntactic representations built from binary relations between tokens (or words) labeled with syntactic functions or dependency types. We define such representations as dependency graphs:

9

Definition 2.2. Given a sentence $x = (w_1, \ldots, w_n)$ and a set $R = \{r_0, r_1, \ldots r_m\}$ of dependency types, a *dependency graph* for a sentence x is a labeled directed graph $G = (V, E, L)$, where:

1. $V = \mathbf{Z}_{n+1} = \{0, 1, 2, \ldots, n\}$

2. $E \subseteq V \times V$

3. $L : E \to R$

A dependency graph consists of a set V of nodes, where a node is a non-negative integer (including n). Every positive node has a corresponding token in the sentence x and we will use the term *token node* for these nodes (i.e., the token w_i corresponds to the token node i). In addition, there is a special root node 0, which is the root of the dependency graph and has no corresponding token in the sentence x. Furthermore, the set V^+ denotes the set of token nodes, i.e., $V^+ = V - \{0\}$. There is a practical advantage in using position indices instead of word forms to represent tokens (Maruyama 1990), which allows the use of the arithmetic relation $<$ to order the nodes, and ensures that every token has a unique node in the graph.

An arc $(i, j) \in E$ connects two nodes i and j in the graph and represents a dependency relation where i is the head and j is the dependent. The notation $i \to j$ will be used for the pair $(i, j) \in E$ and $i \to^* j$ for the reflexive and transitive closure, i.e., $i \to^* j$ if and only if there is a path of zero or more arcs connecting i to j. Finally, the function L labels every arc $i \to j$ with a dependency type $r \in R$ and an arc with a label r will be denoted $i \xrightarrow{r} j$.

To be able to construct a dependency graph using a parsing algorithm, we usually have to define some basic constraints that a graph must satisfy.

Definition 2.3. A dependency graph G is *well-formed* if and only if the following constraints hold:

1. **Root**: The node 0 is a root, i.e., there is a node i such that $i \to 0$.

2. **Connectedness**: G is weakly connected, i.e., for every node i there is some node j such that $i \to j$ or $j \to i$.

3. **Single-Head**: Each node has at most one head, i.e., if $i \to j$ then there is no node k such that $k \neq i$ and $k \to j$.

4. **Acyclicity**: G is acyclic, i.e., if $i \to j$ then not $j \to^* i$.

5. **Projectivity**: G is projective, i.e., if $i \to j$ then $i \to^* k$, for every node k such that $i < k < j$ or $j < k < i$.

10

A special root node makes it easier to comply with the second constraint **Connectedness**, since it is always possible to hook up any node to the root and in that way the graph will always be connected. Furthermore, with a root node we always know the entrance to the graph. The third constraint **Single-Head** (sometimes called uniqueness) is commonly assumed in dependency grammar, although Hudson (1984) allows multiple heads to capture certain transformational phenomena, where a single token is connected to more than one position in the sentence. The fourth constraint **Acyclicity** together with first three constraints entail that the graph is a rooted tree. These assumptions make it simpler to construct parsing algorithms that build dependency trees automatically.

The last constraint **Projectivity** is more controversial and most dependency grammars allow non-projective graphs, because non-projective representations are able to capture non-local dependencies. There exists several treebanks that contain non-projective structures such as the Prague Dependency Treebank of Czech (Hajič et al. 2001) and the Danish Dependency Treebank (Kromann 2003). We will assume the constraint **Projectivity** here because the parsing algorithms used in this thesis are limited to projective structures and the treebanks used only contain projective structures. Moreover, when dealing with non-projective data, it is possible to projectivize the training data and recover non-projective dependencies by applying an inverse transformation after parsing in a post-processing step (Nivre and Nilsson 2005).

Figure 2.1 shows a labeled projective dependency graph for a Swedish sentence, where each word of the sentence is tagged with its part-of-speech and each arc labeled with a dependency type.[2]

2.3 Inductive Dependency Parsing

The framework of inductive dependency parsing, as characterized by Nivre (2006), is based on three essential elements:

1. Deterministic parsing algorithms for building dependency graphs (Kudo and Matsumoto 2002; Yamada and Matsumoto 2003; Nivre 2003)

2. History-based feature models for predicting the next transition from one parser configuration to another (Black et al. 1992; Magerman 1995; Ratnaparkhi 1997; Collins 1999)

---

[2]The dependency types used in Figure 2.1 are described in section 4.1.1.

11

3. Discriminative learning methods to map histories to transitions (Veenstra and Daelemans 2000; Kudo and Matsumoto 2002; Yamada and Matsumoto 2003; Nivre et al. 2004)

In this section we will discuss these three elements. Section 2.3.1 presents two deterministic dependency-based parsing algorithms. Section 2.3.2 describes how history-based models can be used for predicting the next transition from one parser configuration to another. Finally, Section 2.3.3 explains how we can use discriminative learning methods for inducing a classifier that maps parser configurations to transitions, including a brief description of the two learning method used in the experiments: SVM and MBL.

2.3.1 Deterministic Dependency Parsing

Mainstream approaches to data-driven text parsing are based on nondeterministic parsing techniques, but the disambiguation can be performed deterministically, using a greedy parsing algorithm that approximates a globally optimal solution by making a sequence of locally optimal choices (see section 2.4 for more details of related work in this area). The experiments in Chapter 4 will use two parsing algorithms, called *Nivre's algorithm* and *Covington's algorithm*, and both algorithms come in two versions. We begin by defining parser configurations that can be used by both algorithms, following Nivre (2006):

Definition 2.4. Given a set $R = \{r_0, r_1, \ldots, r_m\}$ of dependency types and a sentence $x = (w_1, \ldots, w_n)$, a *parser configuration* for x is a quintuple $c = (\sigma, \tau, \upsilon, h, d)$, where:

1. σ is a stack of partially processed token nodes i ($1 \leq i \leq j$ for some $j \leq n$).

2. τ is a list of remaining input token nodes i ($k \leq i \leq n$ for some $k > j$).

3. υ is a stack of token nodes i occurring between the token on top of the stack σ_j and the next input token τ_k ($j < i < k$).

4. $h : V_x^+ \rightarrow V_x$ is a head function from token nodes to nodes.

5. $d : V_x^+ \rightarrow R$ is a label function from token nodes to dependency types.

6. For every token node $i \in V_x^+$, $d(i) = r_0$ only if $h(i) = 0$.

The definition of a parser configuration introduces three data structures: a stack σ, a list τ and a stack υ. The first two data structures (the stack σ and the list τ) are included in the definition of Nivre (2006). Here the definition

12

is extended with a stack υ, which we call the *context stack* and which is used by Covington's algorithm. In order to define the parsing algorithms later in this section, we will represent all three data structures as lists. To be able to use individual components in these lists, we will use $j|\tau$ to represent a list of input tokens with head j and tail τ, while $\sigma|i$ and $\upsilon|i$ represent stacks with the top i and tail σ and υ. An empty stack/list is represented by ϵ.

The symbols V_x^+ and V_x are used to indicate that V^+ and V are the nodes for the sentence x. The head function h defines the partially built dependency graph. For every token node i there is a syntactic head $h(i) = j$. If the token node i is not yet attached to a head, the special root node $h(i) = 0$ is used.

Finally, the label function d labels the partially built dependency structure, where every token node i is assigned a dependency type r_j using the label function $d(i) = r_j$ ($d(i) = r_0$ is used for token nodes that are not yet attached). We establish a connection between parser configurations and dependency graphs in the following way (Nivre 2006):

Definition 2.5. A parser configuration $c = (\sigma, \tau, \upsilon, h, d)$ for x defines the dependency graph $G_c = (V_x, E_c, L_c)$, where:

1. $E_c = \{(i, j) \,|\, h(j) = i\}$

2. $L_c = \{((i, j), r) \,|\, h(j) = i, d(j) = r\}$

For the functions h and d, we will use the notation $f[x \mapsto y]$; if $f(x) = y'$, then $f[x \mapsto y] = f - \{(x, y')\} \cup \{(x, y)\}$.

Definition 2.6. A parser configuration c for the sentence $x = (w_1, \ldots, w_n)$ is *initial* if and only if it has the form $c = (\epsilon, (1, \ldots, n), \epsilon, h_0, d_0)$, where:

1. $h_0(i) = 0$ for every $i \in V_x^+$.

2. $d_0(i) = r_0$ for every $i \in V_x^+$.

When the parser begins to parse a sentence, the two stacks σ and υ are empty and all the token nodes of the sentence are in the list τ. In the beginning, all token nodes are dependents of the special root node 0 and labeled with the special label r_0. The parser terminates the parsing of a sentence when the following condition is met:

Definition 2.7. A parser configuration c for the sentence $x = (w_1, \ldots, w_n)$ is *terminal* if and only if it has the form $c = (\sigma, \epsilon, \upsilon, h, d)$ (for arbitrary σ, υ, h and d).

13

The parser processes the input left-to-right and terminates whenever the list of input tokens is empty. The set C will denote all possible configurations and C^n the set of non-terminal configurations, i.e., any configuration $c = (\sigma, \tau, \upsilon, h, d)$ where $\tau \neq \epsilon$. A *transition* from a non-terminal configuration to a new configuration is a partial function $t : C^n \to C$.

We will define a transition system for each version of the algorithms, which is nondeterministic. Hence, there will be more than one transition applicable to a given configuration. An *oracle* $o : C^n \to (C^n \to C)$ is used to overcome this nondeterminism (Kay 2000). For each nondeterministic choice point the parsing algorithm will ask the oracle to predict the next transition. In this section we will consider the oracle as a black box, which always knows the correct transition. In section 2.3.2, we will see that we can approximate this oracle by inducing a classifier.

Nivre's algorithm. This parsing algorithm was first proposed for unlabeled dependency parsing by Nivre (2003) and was extended to labeled dependency parsing by Nivre et al. (2004). A sentence $x = (w_1, \ldots, w_n)$ is parsed by the algorithm PARSE-NIVRE in the following way:

PARSE-NIVRE$(x = (w_1, \ldots, w_n))$
1    $c \leftarrow (\epsilon, (1, \ldots, n), \epsilon, h_0, d_0)$
2   **while** $c = (\sigma, \tau, \upsilon, h, d)$ is not terminal
3   **if** $\sigma = \epsilon$
4            $c \leftarrow$ SHIFT(c)
5   **else**
6            $c \leftarrow [o(c)](c)$
7    $G \leftarrow (V_x, E_c, L_c)$
8   **return** G

The algorithm will perform the SHIFT transition if the stack is empty and otherwise let the oracle o predict the next transition $o(c)$ as long as the parser remains in a non-terminal configuration $c \in C^n$. The SHIFT transition pushes the next input token i onto the stack σ. When the terminal configuration is reached the dependency graph is returned.

The algorithm comes in two versions with two transition systems: an *arc-eager* and an *arc-standard* version. The arc-eager version uses four transitions, two of which are parameterized by a dependency type $r \in R$. The transition system updates the parser configuration as follows (Nivre 2006):

Definition 2.8. For every $r \in R$, the following transitions are possible:

1. SHIFT:
   $(\sigma, i | \tau, \epsilon, h, d) \to (\sigma | i, \tau, \epsilon, h, d)$

14

2. REDUCE:
   $(\sigma|i, \tau, \epsilon, h, d) \rightarrow (\sigma, \tau, \epsilon, h, d)$
   if $h(i) \neq 0$

3. RIGHT-ARC(r):
   $(\sigma|i, j|\tau, \epsilon, h, d) \rightarrow (\sigma|i|j, \tau, \epsilon, h[j \mapsto i], d[j \mapsto r])$
   if $h(j) = 0$

4. LEFT-ARC(r):
   $(\sigma|i, j|\tau, \epsilon, h, d) \rightarrow (\sigma, j|\tau, \epsilon, h[i \mapsto j], d[i \mapsto r])$
   if $h(i) = 0$

The transition SHIFT (SH) shifts (pushes) the next input token i onto the stack σ. This is the correct action when the head of the next word is positioned to the right of the next word or the next word is a root. The transition REDUCE (RE) reduces (pops) the token i on top of the stack σ. It is important to ensure that the parser does not pop the top token if it has not been assigned a head, since it will otherwise be left unattached.

The RIGHT-ARC transition (RA) adds an arc from the token i on top of the stack σ to the next input token j, i.e., $i \xrightarrow{r} j$ and involves pushing j onto the stack. Finally, the transition LEFT-ARC (LA) adds an arc from the next input token j to the token i on top of the stack σ, i.e., $j \xrightarrow{r} i$ and involves popping i from the stack. This transition is only allowed when the top token i on the stack has previously received an arc to the special root node 0. We make use of the assumption of projectivity because we know that the top token i cannot have any more left and right dependents and therefore it can be popped.

Nivre's arc-eager algorithm is guaranteed to terminate after at most $2n$ transitions, given a sentence of length n (Nivre 2003). Furthermore, it always produces a dependency graph that is acyclic and projective. The correct transition sequence for the Swedish sentence shown in Figure 2.1 using Nivre's arc-eager algorithm is as follows:

15

$$
\begin{array}{lll}
 & & (\epsilon,\ (1,\ldots,6),\ \epsilon,\ h_0,\ d_0) \\
\text{D} & \mathbf{SH}\!\to & ((1),\ (2,\ldots,6),\ \epsilon,\ h_0,\ d_0) \\
\text{N} & \mathbf{LA(SUB)}\!\to & (\epsilon,\ (2,\ldots,6),\ \epsilon,\ h_1 = h_0[1 \mapsto 2],\ d_1 = d_0[1 \mapsto \mathbf{SUB}]) \\
\text{D} & \mathbf{SH}\!\to & ((2),\ (3,\ldots,6),\ \epsilon,\ h_1,\ d_1) \\
\text{N} & \mathbf{RA(ADV)}\!\to & ((2,3),\ (4,5,6),\ \epsilon,\ h_2 = h_1[3 \mapsto 2], d_2 = d_1[3 \mapsto \mathbf{ADV}]) \\
\text{N} & \mathbf{RE}\!\to & ((2),\ (4,\ldots,6),\ \epsilon,\ h_2,\ d_2) \\
\text{N} & \mathbf{RE(ADV)}\!\to & ((2,4),\ (5,6),\ \epsilon,\ h_3 = h_2[4 \mapsto 2],\ d_3 = d_2[4 \mapsto \mathbf{ADV}]) \\
\text{N} & \mathbf{RA(PR)}\!\to & ((2,4,5),\ (6),\ \epsilon,\ h_4 = h_3[5 \mapsto 4],\ d_4 = d_3[5 \mapsto \mathbf{PR}]) \\
\text{N} & \mathbf{RE}\!\to & ((2,4),\ (6),\ \epsilon,\ h_4,\ d_4) \\
\text{N} & \mathbf{RE}\!\to & ((2),\ (6),\ \epsilon,\ h_4,\ d_4) \\
\text{N} & \mathbf{RA(IP)}\!\to & ((2,6),\ \epsilon,\ \epsilon, h_5 = h_4[6 \mapsto 2],\ d_5 = d_4[6 \mapsto \mathbf{IP}]) \\
\end{array}
$$

The first row presents the initial parser configuration with an empty stack and $h_0(i) = 0$ and $d_0(i) = r_0$ for every node $i \in V$. The second row shows the parser configuration after the shift transition has been executed. The left column tells us if the transition is deterministic (D) or nondeterministic (N), in other words if the oracle o is used or not. For example, the second row can only be a shift transition because the stack is empty (D) and the third row is a nondeterministic transition (N).

The arc-standard version uses a strict bottom-up processing as in traditional shift-reduce parsing. The algorithm by Kudo and Matsumoto (2002), Yamada and Matsumoto (2003) and Cheng et al. (2005a) uses the arc-standard strategy, but also allows multiple passes over the input.

The arc-standard version uses a transition system similar to the arc-eager version, but has only three transitions SHIFT, LEFT-ARC and RIGHT-ARC (no REDUCE). The first two transitions, SHIFT and LEFT-ARC, are applied in exactly the same way as for the arc-eager version. The transition system is defined as follows:

1. SHIFT:
   $(\sigma, i|\tau, \epsilon, h, d) \to (\sigma|i, \tau, \epsilon, h, d)$

2. RIGHT-ARC(r):
   $(\sigma|i, j|\tau, \epsilon, h, d) \to (\sigma, i|\tau, \epsilon, h[j \mapsto i], d[j \mapsto r])$
   if $h(j) = 0$

3. LEFT-ARC(r):
   $(\sigma|i, j|\tau, \epsilon, h, d) \to (\sigma, j|\tau, \epsilon, h[i \mapsto j], d[i \mapsto r])$
   if $h(i) = 0$

Instead of pushing the next token j onto the stack σ, RIGHT-ARC moves the topmost token i on the stack back to the list of remaining input tokens τ,

16

where it replaces the token j as the next token. The transition sequence for the same sentence using Nivre's arc-standard algorithm:

$$
\begin{array}{lll}
 & & (\epsilon,\ (1,\ldots,6),\ \epsilon,\ h_0,\ d_0) \\
\text{D} & \mathbf{SH}\rightarrow & ((1),\ (2,\ldots,6),\ \epsilon,\ h_0,\ d_0) \\
\text{N} & \mathbf{LA(SUB)}\rightarrow & (\epsilon,\ (2,\ldots,6),\ \epsilon, h_1 = h_0[1 \mapsto 2],\ d_1 = d_0[1 \mapsto \mathbf{SUB}]) \\
\text{D} & \mathbf{SH}\rightarrow & ((2),\ (3,\ldots,6),\ \epsilon,\ h_1,\ d_1) \\
\text{N} & \mathbf{RA(ADV)}\rightarrow & (\epsilon,\ (2,4,\ldots,6),\ \epsilon, h_2 = h_1[3 \mapsto 2],\ d_2 = d_1[3 \mapsto \mathbf{ADV}]) \\
\text{D} & \mathbf{SH}\rightarrow & ((2),\ (4,\ldots,6),\ \epsilon,\ h_2,\ d_2) \\
\text{N} & \mathbf{SH}\rightarrow & ((2,4),\ (5,\ldots,6),\ \epsilon,\ h_2,\ d_2) \\
\text{N} & \mathbf{RA(PR)}\rightarrow & ((2),\ (4,6),\ \epsilon, h_3 = h_2[5 \mapsto 4],\ d_3 = d_2[5 \mapsto \mathbf{PR}]) \\
\text{N} & \mathbf{RA(ADV)}\rightarrow & (\epsilon,\ (2,6),\ \epsilon, h_4 = h_3[4 \mapsto 2],\ d_4 = d_3[4 \mapsto \mathbf{ADV}]) \\
\text{D} & \mathbf{SH}\rightarrow & ((2),\ (6),\ \epsilon,\ h_4,\ d_4) \\
\text{N} & \mathbf{RA(IP)}\rightarrow & (\epsilon,\ (2),\ \epsilon, h_5 = h_4[6 \mapsto 2],\ d_5 = d_4[6 \mapsto \mathbf{IP}]) \\
\text{D} & \mathbf{SH}\rightarrow & ((2),\ \epsilon,\ \epsilon,\ h_5,\ d_5)
\end{array}
$$

We can see that the transitions are performed in another order, for instance the RIGHT-ARC(PR) is executed before RIGHT-ARC(ADV), compared to the arc-eager version.

Covington's algorithm. Covington (2001) proposes several incremental parsing algorithms for dependency parsing. Two of the algorithms are the *projective* algorithm and the *exhaustive left-to-right search* algorithm. The first algorithm uses a *headlist* with words that do not yet have heads and a *wordlist* with all words encountered so far. We will not use these two data structures; instead we will describe these two algorithms by using the data structures defined by the parser configuration: the stacks σ and υ, and the list τ. Actually, we will regard these two algorithms as one algorithm with two transition systems or as two versions of the same algorithm. We will call the second version the *unrestricted*, because it allows dependency graphs that are non-projective and cyclic. Both versions have quadratic complexity, since they proceed by trying to link each new token to each preceding token. It is also possible to define other versions. For example, a version that conforms to the **Acyclicity** requirement but allows non-projective graphs, but this will not be done in this thesis. The adapted version of Covington's algorithm is described as follows:

17

$\textsc{Parse-Covington}(x = (w_1, \ldots, w_n))$

1	$c \leftarrow (\epsilon, (1, \ldots, n), \epsilon, h_0, d_0)$
2	**while** $c = (\sigma, \tau, \upsilon, h, d)$ is not terminal
3	$\quad$ $\textsc{Done} \leftarrow false$
4	$\quad$ **while** $\sigma \neq \epsilon$ **and** $\neg\textsc{Done}$
5	$\quad\quad$ $c \leftarrow [o(c)](c)$
6	$\quad$ **while** $\upsilon \neq \epsilon$
7	$\quad\quad$ $\textsc{Push}(\textsc{Pop}(\upsilon), \sigma)$
8	$\quad$ $\textsc{Push}(\textsc{First}(\tau), \sigma)$
9	$G \leftarrow (V_x, E_c, L_c)$
10	**return** G

The algorithm begins by initializing the configuration with two empty stacks and all token nodes in the list τ, in the same way as Nivre's algorithm. As long as the parser remains in a non-terminal configuration, it will first iterate as long as the stack σ is not empty or the flag $\textsc{Done}$ is false. The $\textsc{Done}$ flag is only used by the projective version to indicate that it can proceed to the next token without an empty stack. Before it can proceed with the next input token, the algorithm must move back all unattached tokens in the context stack υ to the stack σ. The $\textsc{Push}$ function pushes a token onto a stack and the $\textsc{Pop}$ function pops a token from a stack. Finally, the next input token is pushed onto the stack σ, using the function $\textsc{First}$ to retrieve the first token in a list.

The unrestricted version uses three transitions and these are defined in the following way:

1. $\textsc{Reduce}$:
   $(\sigma|i, \tau, \upsilon, h, d) \rightarrow (\sigma, \tau, \upsilon|i, h, d)$

2. $\textsc{Right-Arc}(r)$:
   $(\sigma|i, j|\tau, \upsilon, h, d) \rightarrow (\sigma, j|\tau, \upsilon|i, h[j \mapsto i], d[j \mapsto r])$

3. $\textsc{Left-Arc}(r)$:
   $(\sigma|i, j|\tau, \upsilon, h, d) \rightarrow (\sigma, j|\tau, \upsilon|i, h[i \mapsto j], d[i \mapsto r])$

All three transitions move the top token of the stack σ to the stack υ. The $\textsc{Right-Arc}$ and $\textsc{Left-Arc}$ transitions in addition add an arc $i \xrightarrow{r} j$ or an arc $j \xrightarrow{r} i$, respectively.

The projective version makes use of the fact that it should build a projective graph, which allows the algorithm to continue with the next input token

18

without exploring all combinations that could make the graph non-projective. The transition system is redefined as follows:

1. REDUCE:
   $(\sigma|i, j|\tau, \upsilon, h, d) \rightarrow (\sigma, j|\tau, \upsilon|i, h, d)$ DONE $\leftarrow$ *true*

2. RIGHT-ARC(r):
   $(\sigma|i, j|\tau, \upsilon, h, d) \rightarrow (\sigma, j|\tau, \upsilon|i, h[j \mapsto i], d[j \mapsto r])$ DONE $\leftarrow$ *true*
   if $h(j) = 0$

3. LEFT-ARC(r):
   $(\sigma|i, j|\tau, \upsilon, h, d) \rightarrow (\sigma, j|\tau, \upsilon, h[i \mapsto j], d[i \mapsto r])$
   if $h(i) = 0$

The REDUCE transition is exactly the same as for the unrestricted version except that it sets the DONE flag to true, in order to indicate that all the remaining tokens in stack σ cannot be linked to the token j, since this would produce a non-projective graph. The RIGHT-ARC transition makes use of the fact that the arc $i \xrightarrow{r} j$ covers the tokens between the top token and the next token; to prevent that the graph becomes non-projective the top token i of stack σ is popped and then pushed onto the context stack υ and the flag DONE is assigned the value true, for the same reason as in the REDUCE transition. The LEFT-ARC transition adds an arc $j \xrightarrow{r} i$; because i cannot be linked to another token it is popped from the stack σ.

2.3.2 History-Based Models

In section 2.3.1 we defined a set C of possible parser configurations and for each version of the parsing algorithm we defined a transition system is non-deterministic. Furthermore, we introduced an oracle $o : C^n \rightarrow (C^n \rightarrow C)$, which the parsing algorithm uses to get the correct transition. If it is possible to derive the correct transitions from syntactically annotated sentences, we can use these as training data to approximate such an oracle through inductive learning. In other words, we define a one-to-one mapping from an input string x and a dependency graph G to a sequence of transitions $S = (t_1, \ldots, t_m)$ such that S uniquely determines G. A transition t_i is dependent on all previously made transitions $(t_1, \ldots, t_{i-1})$ and all available information about these transitions, called the *history*. The history $H_i = (t_1, \ldots, t_{i-1})$ corresponds to some partially built structure and we also include static properties that are kept constant during the parsing of a sentence, such as word form and part-of-speech of a token.

19

The basic idea is thus to train a classifier that approximates an oracle given that a treebank is available. We will call the approximated oracle a *guide* (Boullier 2003), because the guide does not guarantee that the transition is correct. The history $H_i = (t_1, \ldots, t_{i-1})$ contains complete information about all previous transitions. All this information is intractable for training a classifier. Instead we can use history-based feature models for predicting the next transition. History-based feature models were first introduced by Black *et al.* (1992) and have been used extensively in data-driven parsing (Magerman 1995; Ratnaparkhi 1997; Collins 1999). To make it tractable the history H_i is replaced by a feature vector defined by a feature model $\Phi = (\phi_1, \ldots, \phi_p)$, where each feature ϕ_i is a function that identifies some significant property of the history H_i and/or the input string x. To simplify notation we will write $\Phi(H_i, x)$ to denote the application of the feature vector $(\phi_1, \ldots, \phi_p)$ to H_i and x, i.e., $\Phi(H_i, x) = (\phi_1(H_i, x), \ldots, \phi_p(H_i, x))$.

At learning time the parser derives the correct transition by using an oracle function o applied to gold standard treebank. For each transition it provides the learner with a training instance $\Phi((H_i, x), t_i)$, where $\Phi(H_i, x)$ is a current vector of feature values and t_i is the correct transition. A set of training instances I is then used by the learner to induce a parser model, by using a supervised learning method.

At parsing time the parser uses the parser model, as a guide, to predict the next transition and now the vector of feature values $\Phi(H_i, x)$ is the input and the transition t_i is the output of the guide. Section 2.3.3 describes how we can train a classifier that makes this prediction.

2.3.3 Discriminative Learning Methods

The learning problem is to induce a classifier from a set of training instances I relative to a specific feature model Φ by using a learning algorithm. In this section, we will describe two discriminative learning methods, SVM and MBL, that can be used for this classification task.

In general, classification is the task of predicting the class y given a variable x, which can be accomplished by probabilistic methods and it is common to divide these methods into two classes: *generative* and *discriminative*. For generative methods, we use the Bayes rule to obtain $P(y \,|\, x)$ by estimating the joint distribution $P(x, y)$. By contrast, discriminative methods make no attempt to model underlying distributions and instead estimate $P(y \,|\, x)$ directly. We will use two discriminative methods for the learning task: SVM and MBL.

Support Vector Machines. In the last decade, there has been a growing interest in Support Vector Machines (SVM), which were proposed by

20

Vladimir Vapnik at the end of the seventies (Vapnik 1979). SVM is based on the idea that two linearly separable classes, the positive and negative samples in the training data, can be separated by a hyperplane with the largest margin. It has been shown that SVMs give good generalization performance in various research areas, such as face detection (Osuna et al. 1997) and pedestrian detection (Oren et al. 1997). Within natural language processing they have been used extensively in, for example, text categorization (Joachims 1998), chunking (Kudo and Matsumoto 2001) and syntactic parsing (Yamada and Matsumoto 2003).

Given a data set of ℓ instance-label pairs $I = \{(\overrightarrow{x_i}, y_i)\}_{i=1}^{\ell}$, where $x_i \in \mathbb{R}^N$ and $y_i \in \{-1, 1\}$, x_i is a feature vector of the i-th sample, which is represented by an n dimensional vector $\overrightarrow{x_i} = (f_1, \ldots, f_n)$, and y_i is the class label of the i-th sample which belongs to either the positive $(+1)$ or the negative (-1) class. The feature vector $\overrightarrow{x_i}$ will in our case be the feature vector defined by $\Phi(H_i, x)$ and the class label y_i will be the transition t_i, but we need a method that handles multiple class labels (more about that later in this section). The idea is to estimate a vector $\overrightarrow{w}$ and a scalar b, which maximize the distance of any data point from the hyperplane defined by $\overrightarrow{w} \cdot \overrightarrow{x} + b$. The goal of the SVM is to find the solution of the following optimization (Kudo and Matsumoto 2000a; Burges 1998):

$$\begin{array}{ll} \text{Minimize:} & L(w) = \frac{1}{2} \parallel \overrightarrow{w} \parallel^2 \\ \text{Subject to:} & y_i(\overrightarrow{w} \cdot \overrightarrow{x_i} + b) \geq 1 \forall i = 1, \ldots, \ell \end{array} \tag{2.1}$$
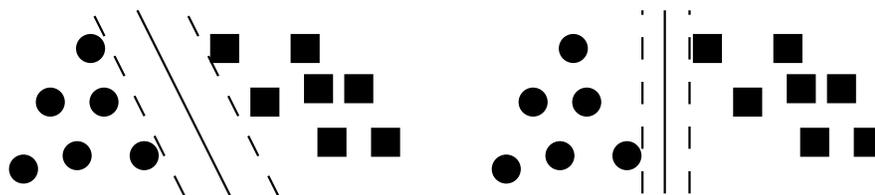

Figure 2.2: A linear Support Vector Machine

In other words, the SVM method tries to find the hyperplane that separates the training data into two classes with the largest margin. Figure 2.2 illustrates two possible hyperplanes, which correctly separate the training data into two classes, and the left hyperplane has the largest margin between the two classes.

21

The data in Figure 2.2 are easy to separate into two classes, but in practice the data may be noisy and therefore not linearly separable. One solution is to allowing some misclassifications by introducing a penalty parameter C, which defines the trade off between the training error and the magnitude of the margin.

SVM can be extended to solve problems that are not linearly separable. The feature vector x_i is mapped to a higher dimensional space by the function ϕ, which makes it possible to carry out non-linear classification. The optimization problem can be rewritten into a dual form, which is done with a so called Kernel function $K(x_i, x_j) \equiv \phi(x_i)^T \phi(x_j)$ (Kudo and Matsumoto 2001; Vapnik 1998). There are many kernel functions, but the most common are:

- polynomial: $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d, \gamma > 0$.

- radial basis function (RBF): $K(x_i, x_j) = exp(-\gamma \parallel x_i - x_j \parallel^2), \gamma > 0$.

- sigmoid: $K(x_i, x_j) = tanh(\gamma x_i^T x_j + r)$.

where γ, r and d denote different kernel parameters (Hsu et al. 2004).

SVM is in its basic form a binary classifier, but many learning problems have to deal with more than two classes. To make SVM handle multi-classification, many binary classifiers are used. For multi-class classification, we can choose between the methods one-against-all and all-against-all. Given that we have n classes, the one-against-all method trains n classifiers to separate each class from the rest and the all-against-all method trains $n(n-1)/2$ classifiers, one for each pair of classes (Vural and Dy 2004). A voting mechanism or some other measure is used to discriminate across all these classifiers to classify a new instance.

Memory-Based Learning. Memory-based learning (MBL) and classification is based on the assumption that a cognitive learning task to a high degree depends on direct experience and memory, rather than extraction of an abstract representation. MBL has been used for many language learning tasks, such as part-of-speech tagging (Cardie 1993; Daelemans et al. 1996), semantic role labeling (Van den Bosch et al. 2004; Kouchnir 2004) and syntactic parsing (Nivre et al. 2004).

MBL is a lazy method and is based on two fundamental principles: learning is storing experiences in memory, and solving a new problem is achieved by reusing solutions from previously solved problems that are similar to the new problem. The idea during training for MBL is to collect the values of different features from the training data together with the correct class

22

(Daelemans and Van den Bosch 2005). MBL generalizes by applying a similarity metric without abstracting or eliminating low-frequency events. This similarity metric can be seen as an implicit smoothing mechanism for rare events. Daelemans and colleagues have shown that it may be harmful to eliminate rare events in the training data for language learning tasks (Daelemans et al. 2002), because it is very difficult to discriminate noise from valid exceptions.

The n feature-values are mapped into an n-dimensional space, where each feature vector from the training data with its corresponding class is a point in this space. The task at decision time is to find the nearest neighbor(s) in this n-dimensional space and return a category based on the k nearest neighbor(s). The way this search is performed can be varied in many different ways.

The *Overlap metric* is one of the most basic metrics and uses the distance $\Delta(X, Y)$ between two patterns X and Y, which are represented as n features:

$$\Delta(X, Y) = \sum_{i=1}^{n} w_i \delta(x_i, y_i) \qquad (2.2)$$

where w_i is a weight for feature i, and the function $\delta(x_i, y_i)$ is the distance per feature and will be 0 if $x_i = y_i$, otherwise 1. The weight w_i can be calculated by a variety of methods, e.g. *Information Gain* (IG), which measures each feature's contribution to our knowledge with respect to the target class.

A variation of the Overlap metrics is the more sophisticated Modified Value Difference Metric (MVDM), introduced by Cost and Salzberg (1993), which estimates the distance between two values of a feature by considering their cooccurrence with the target classes. However, this metric is more sensitive to sparse data.

2.4 Related Work

During the last decades, there has been a great interest in data-driven methods for various natural language processing tasks. Data-driven approaches to syntactic parsing were first developed during the 90s for constituency-based representations. The standard approaches are based on nondeterministic parsing techniques, usually involving some kind of dynamic programming, in combination with generative probabilistic models that provide an n-best ranking of the set of candidate analyses derived by the parser. The most well-known parsers based on these techniques are the parser of Collins (1997, 1999) and the parser of Charniak (2000). Discriminative learning methods have been used to enhance these parsers by reranking the analyses output

23

by the parser (Johnson et al. 1999; Collins and Duffy 2005; Charniak and Johnson 2005).

Dependency-based data-driven parsing was first introduced by Eisner (1996), who used a probabilistic parser to assign both part-of-speech tags and an unlabeled (bare-bone) dependency structure simultaneously. The parser of Eisner was evaluated on the Wall Street Journal section of the Penn treebank and the study showed that a generative model performs significantly better than other models based on, for example, lexical affinity and sense tagging. In later work, Eisner enhanced his approach by defining the notion of *bilexical grammar* (Eisner 1999). More recently, McDonald et al. (2005) have come up with an alternative approach to dependency parsing, which uses a modified version of the Eisner's algorithm (Eisner 1999) and an online large-margin learning algorithm. This methodology has been shown to give state-of-the-art performance without any language specific enhancements for both English and Czech.

Collins's parser has been adapted to handle dependency-based structures and the evaluation shows that on the Wall Street Journal data (English) it gives an unlabeled dependency accuracy of 91% and for Czech, which is a highly inflected language with relatively free word order, it achieves an accuracy of 82% (Collins et al. 1999).

Another school proposes that dependency parsing can be performed deterministically, guided by a classifier trained on gold standard derivations from a treebank (the technique used in this thesis). It was first used for unlabeled dependency parsing by Kudo and Matsumoto (2000a, 2002) (for Japanese) and Yamada and Matsumoto (2003) (for English) using SVM. The parsing algorithm uses a variation of shift-reduce parsing with three possible parse actions: SHIFT, RIGHT and LEFT. The two latter parse actions add a dependency relation between two target nodes, which are two neighboring tokens. The parse action SHIFT moves the focus to the right in the input string, which results in two new target nodes. The worst-case time complexity of this approach is $O(n^2)$, but the worst-case rarely occurs in practice. Cheng *et al.* (2005a) have recently used this methodology to parse Chinese with state-of-the-art performance.

The deterministic dependency parsing was extended to handle labeled dependency structures by Nivre et al. (2004) (for Swedish) and Nivre and Scholz (2004) (for English). Sagae and Lavie (2005) have investigated a deterministic approach with good results based on lexicalized phrase structure. The parsing framework of Yamada and Matsumoto is very similar to the Nivre's arc-standard version (see section 2.3.1, but the difference is that Nivre's approach constructs labeled dependency graphs and Yamada and Matsumoto allows multiple passes over the input.

24

Chapter 3

MaltParser

This chapter describes a dependency parser called MaltParser (Nivre et al. 2006) and its underlying architecture. The theoretical background of this parser is the framework of inductive dependency parsing introduced by Nivre (2006) and briefly described in section 2.3 with some extensions. The development of MaltParser has been joint work with especially Joakim Nivre, but also Jens Nilsson. Version 0.4 of MaltParser is the version described in this thesis. The chapter is divided into two sections. Section 3.1 explains the underlying architecture and section 3.2 describes the implementation of MaltParser version 0.4 in more detail.

3.1 Architecture

We propose an architecture for labeled dependency parsing with a strict modularization of parsing algorithms, feature models and learning methods, thereby giving maximum flexibility in the way these components can be varied independently of each other. The architecture can be seen as a data-driven parser-generator framework given that a treebank is available. Whereas a traditional parser-generator maps a grammar to a parser, a data-driven parser-generator maps a treebank to a parser. With different settings for parsing algorithms, feature models and learners the parser-generator will construct different parsers. The idea is to give the user the flexibility to experiment with the components in a more convenient way, although there are still dependencies between components, in the sense that not all combinations will perform well with respect to accuracy and efficiency.

The design of the architecture deals also with the fact that the parser can be executed in two different phases: *the learning phase* and *the parsing phase.* In the learning phase, the system uses a treebank to learn a model; in the parsing phase, it takes a previously learnt model and uses this to parse new and unseen data. Although these two tasks have a different structure, they often involve similar or even identical subproblems. For instance, learning normally requires the construction of a gold standard parse derivation in

25
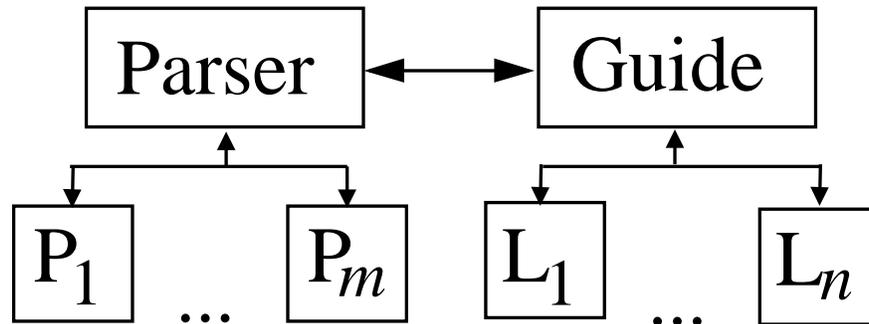
Figure 3.1: Architecture

order to estimate the parameters of a model, which is exactly the same kind of derivation that is constructed during parsing. One advantage of finding joint subproblems for the two phases is that we can reuse components across phases.

The architecture consists of two main components:

1. **Parser**: The Parser derives the dependency graph G for a sentence x (during both learning and parsing) using a deterministic parsing algorithm. The parser component contains subcomponents $P_1, \ldots, P_m$, where each subcomponent defines a parsing algorithm.

2. **Guide**: The Guide extracts feature vectors from the current state of the system (according to the specified model) and passes data between a learner and the parser. A learner is responsible for inducing a model at learning time and for using this model to guide the parser to make decisions at all nondeterministic choice points during the parsing phase. The guide component can use several learners $L_1, \ldots, L_n$. Usually a learner is an interface to a software package that implements a machine learning algorithm.

Figure 3.1 illustrates the architecture. In addition to the two main components, with their subcomponents, the framework includes input/output components and an overall control structure. These components are responsible for reading a text $T = (x_1, \ldots, x_n)$ and invoking the Parser component for each sentence $x_i \in S$. When the Parser has constructed the dependency graph G, the dependency graph should be output in a suitable format. Sections 3.1.1 and 3.1.2 describe the main components in more detail.

26

3.1.1 Parser

The Parser is responsible for the derivation of a well-formed dependency graph G for a sentence $x = w_1, \ldots, w_n$, given a set $R = \{r_0, r_1, \ldots, r_m\}$ of dependency types (where r_0 is a special symbol for dependents of the root), as defined in section 2.2. To perform this derivation, we need a parsing algorithm that can map a dependency graph into a sequence of transitions $S = (t_1, \ldots, t_m)$ during learning time by using the gold standard. Each nondeterministic transition t_i will be passed on to the Guide as an example of a correct transition. During parsing time, the parsing algorithm constructs a dependency graph by asking the Guide for each nondeterministic transition t_i.

Moreover, one of the main ideas of the Parser component is that it should be capable of incorporating several parsing algorithms, and allow the user of the system to choose a suitable algorithm for a specific purpose. Not all parsing algorithms are applicable for this architecture and therefore we need to define the restrictions that the parsing algorithm must fulfill:

1. It must be able to use parser configurations $c = (\sigma, \tau, v, h, d)$ of the form defined in Definition 2.4 for each sentence x.

2. It must only assign a dependency relation between the token on top of the stack σ and the next input token in the list τ.

3. It must be deterministic, in the sense that it always derives a single analysis for each sentence x. However, the algorithm can be nondeterministic in its individual choice points such as adding a dependency arc between two tokens or shifting a new token onto its stack. This implies that the algorithm must be able to break down the parsing problem into a sequence of transitions $S = (t_1, \ldots, t_m)$.

4. It must be possible to define an oracle given a treebank $T_t = (x_1, \ldots, x_n)$ in the learning phase. In this way, the algorithm derives the correct transition at all nondeterministic choice points using the oracle function. This function defines the mapping between the dependency graph G and the transition sequences S such that S uniquely determines the dependency graph G.

5. It must define a set of permissible transitions for each configuration c and check that it does not perform an illegal transition. For example, if the stack σ is empty then it is not possible to pop the stack. To comply with the requirement of robustness, it must therefore provide a default transition according to the current state of the system if an illegal transition is proposed by the guide.

27

3.1.2 Guide

The Guide is responsible for constructing a set of training instances I for a learner subcomponent during the learning phase, and for passing a learner's predictions to the Parser during the parsing phase.

At learning time, the Guide constructs one training instance $\Phi((H_i, x), t_i)$ for each transition t_i passed from the Parser, where $\Phi(H_i, x)$ is the current vector of feature values (given the feature model Φ and the current state of the system) and passes this on to a learner. At parsing time, the Guide constructs a feature vector $\Phi(H_i, x)$ for each request from the Parser, sends it to a learner and passes on the predicted transition t_i from a learner to the Parser. In this way, the feature model is completely separated from the Parser, and the currently used learner only has to learn a mapping from feature vectors to transitions, without knowing either how the features are extracted or how the transitions are to be used. Moreover, the feature extraction is performed in exactly the same way during both learning and parsing.

The feature extraction uses the feature model Φ, which is defined in terms of a feature vector $(\phi_1, \ldots, \phi_p)$, where each feature ϕ_i is a function, defined in terms of three simpler functions: an *address function* a_{ϕ_i}, which identifies a specific token in a given parser configuration, an *attribute function* f_{ϕ_i}, which picks out a specific attribute of the token, and a *mapping function* m_{ϕ_i}, which defines equivalence classes of attribute values.

1. For every i, $i \geq 0$, $\sigma[i]$, $\tau[i]$ and $\upsilon[i]$ are address functions identifying the $i+1$th token from the top of the stack σ, the start of the input list τ, and the top of the stack υ, respectively. (Hence, $\sigma[0]$ is the top of the stack, $\tau[0]$ is the next input token and $\upsilon[0]$ is the top of the context stack.)

2. If α is an address function, then $l(\alpha)$ and $r(\alpha)$ are address functions, identifying the left and right string neighbors, respectively, of the token identified by α.

3. If α is an address function, then $h(\alpha)$, $lc(\alpha)$, $rc(\alpha)$, $ls(\alpha)$ and $rs(\alpha)$ are address functions, identifying the head (h), the leftmost child (lc), the rightmost child (rc), the next left sibling (ls) and the next right sibling (rs), respectively, of the token identified by α (according to the partially built dependency graph G).

4. If α is an address function, then $f(\alpha)$ feature functions, identifying a particular attribute of the token identified by α. The part-of-speech (p), word form (w) and dependency type (d) are example of attributes which can be identified (where the dependency type, if any, is given by

28

the partially built dependency graph G). We call p, w and d attribute functions. It is possible to extend with more attribute functions if there are more attributes available in the data.

5. If $f(\alpha)$ is a feature function, then $m(f(\alpha))$ is a feature function if m is a mapping from the range of f to some new value set. For example, a mapping function can be used to restrict the value of a lexical feature to the last n characters of the word form.

Given this feature model $\Phi(H_i, x)$, the Guide will extract a feature vector $(v_1, \ldots, v_p)$ and passes this to the learner, where v_j is the extracted value for the corresponding feature ϕ_j. During learning it also provides the transition t_i; during parsing it instead requests t_i given the feature vector $(v_1, \ldots, v_p)$.

A learner subcomponent, finally, is responsible for inducing a classifier g from the set of training instances I by using a learner L. The *learned function* g is an approximation of the true oracle o. The set of possible transitions is discrete and finite, and therefore we can view this as a *classification* problem. The classifier g is used to predict a transition given a parser state during the parsing phase. In practice, a learner will normally be an interface to a standard machine learning package. A learner must meet the following requirements:

1. It must at learning time be able to induce a model that is stored so that it can be loaded when needed for parsing new data, given a set of training instances I.

2. It must be able to create a model based on only the training instances; no background knowledge is allowed.

3. It must be capable at parsing time to discriminate between several possible transitions so that it returns at most one transition (disambiguation) and be robust in the sense that it always returns at least one transition (robustness), although the parser may use a default transition if the learner fails.

The first two requirements exclude learning systems that cannot store its model offline and systems that depend on background knowledge, for example, knowledge about a specific language. The third requirement ensures that the learner complies with the deterministic approach to parsing. This entails that discriminative learning algorithms are well-suited for this kind of classification task.

29

3.2 Implementation

The architecture described in section 3.1 has been realized in the MaltParser system, which can be applied to a labeled dependency treebank in order to induce a labeled dependency parser for the language represented by the tree-bank. MaltParser is freely available for research and educational purposes.[1] The software is terminal-based and is controlled by an option file or by sup-plying it with appropriate parameters using command line flags. The latter override any settings included in the option file. MaltParser is primarily written in the programming language C with a few parts written in C++. The system can be run in two basic modes, the learning mode LEARN and the parsing mode PARSE. The following algorithm describes the main-loop of MaltParser:

MALTPARSER($T = (x_1, \ldots, x_n)$, $Z = (z_1, \ldots, z_m)$)
1   INITIALIZE(Z)
2   **while** T not empty
3        $x \leftarrow$ GETNEXTSENTENCE(T)
4        $y \leftarrow$ PARSE(x)
5   OUTPUTANALYSIS(x, y)
7   TERMINATE(Z)

The algorithm above is a little bit simplified, and captures only the important aspects. MaltParser takes a text $T = (x_1, \ldots, x_n)$ and a set of settings $Z = (z_1, \ldots, z_m)$. The text T must be sentence segmented, tokenized and part-of-speech tagged; when running in the LEARN mode it must also be syntactically annotated with dependency graphs. The elements of Z are feature-value pairs $z_i = (f_i, v_i)$. For example, (LEARNER, SVM) means that the learner or classifier type used is SVM. Note that the main-loop is the same for the LEARN and the PARSE modes.

The function INITIALIZE initializes all components according to the set-tings Z. Every sentence in T is read from a file into a *buffer* x by the function GETNEXTSENTENCE. The complex data structure buffer contains a list of n tokens, where each token has attributes such as word form, part-of-speech, head and dependency type. Each sentence is parsed by the function PARSE, which assigns an analysis y to the sentence x by updating the head and de-pendency type attributes for each word in the buffer. The invocation of the function PARSE is the same as telling the Parser component to begin parsing the sentence (and derive training instances in the LEARN mode). The anal-ysis y is output to a file together with the sentence x. MaltParser ends by

---

[1]MaltParser can be downloaded from http://www.vxu.se/msi/users/nivre/research/MaltParser.html.

30

calling the function TERMINATE, which deletes all data structures according to the settings Z.

Section 3.2.2 describes the two main components and their subcomponents, called the parser kernel, from the point of view of how they are implemented in the MaltParser system, but we begin with a brief overview of the IO component in section 3.2.1.

3.2.1 Input and Output

MaltParser takes an input file in the Malt-TAB format,[2] where each token is represented on one line, with attribute values being separated by tabs and each sentence separated by a blank line. An example of the Malt-TAB format is shown below for the dependency graph in Figure 2.1:

```
Cykelreglerna    nn.nom    2    SUB
gäller           vb.fin    0    ROOT
också            ab        2    ADV
för              pp        2    ADV
mopedister       nn.nom    4    PR
.                mad       2    IP
```

The first two columns (word form and part-of-speech) are required both during learning and parsing. The last two columns (head and dependency type) are only required during learning. The head is an index to the implicit position of the head in the sentence. For example, the token *Cykelreglerna* has the index 1 and its head *gäller* has the index 2. The part-of-speech and dependency type are mapped to integer values and each part-of-speech and dependency type has its unique integer value. When the Parser component has processed the sentence, the analysis and the sentence are written to a file in the Malt-TAB format or the Malt-XML format. The latter is an XML version of the Malt-TAB format.

3.2.2 Parser Kernel

Figures 3.2 and 3.3 illustrate the parser kernel at parsing and learning time. The MALTPARSER algorithm invokes the Parser component by using the function PARSE. Furthermore, we can also see the communication between the two main components *Parser* and *Guide*. In addition to the functions INITIALIZE and TERMINATE, there are three functions. The RESET function

---

[2]The guidelines for the Malt-TAB format can be found at this web page: http://www.vxu.se/msi/users/nivre/research/MaltXML.html. MaltParser version 0.4 also support the format of the CoNLL-X shared task, but this format will not be described in this thesis.
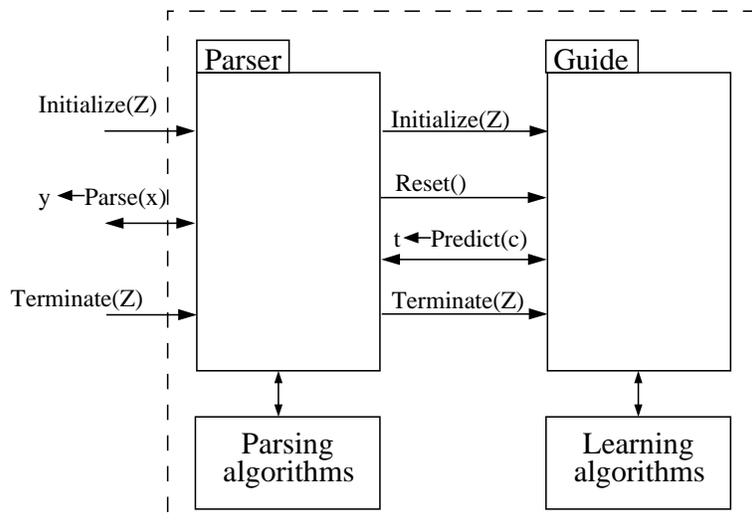
Figure 3.2: The parser kernel at parsing time
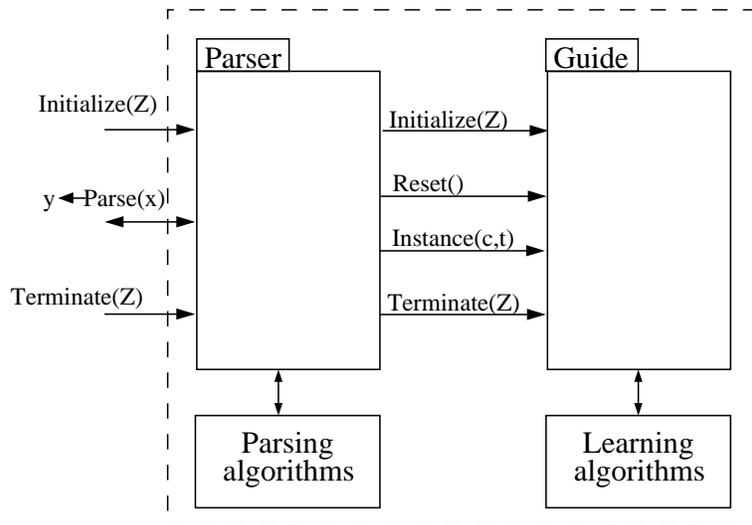
Figure 3.3: The parser kernel at learning time

32
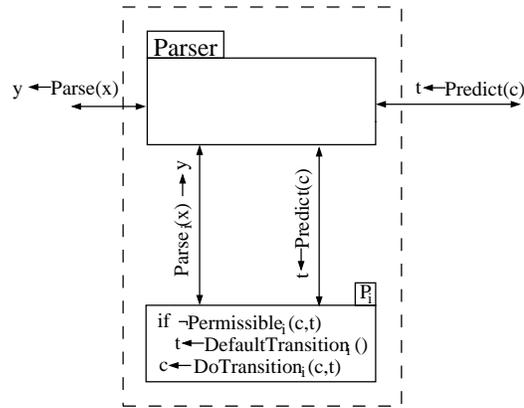
Figure 3.4: The parser component at parsing time

is used to tell the Guide that a new sentence will be parsed and that the Guide should reset its data structures, if necessary. At parsing time, the PREDICT function is used to request a transition t from the Guide based on the parser configuration c. The Guide extracts the feature values according to the feature model and then delegates the request to its learner subcomponent L_i, which derives the transition using a classifier g. At learning time, the only difference at this level of abstraction is that the Parser instead invokes the INSTANCE function, which adds the training instance to the sequence of training instances.

3.2.3 Parser

The Parser component is invoked by the MALTPARSER function to parse a sentence. It has access to the buffer where all token nodes are stored with all their attributes. In addition, it needs a stack σ, a list τ and a context stack v, which contain integer values pointing to token nodes in the buffer. Before parsing the sentence x it resets its data structures: $\sigma = \epsilon$, $\tau = (1, \ldots, n)$ and $v = \epsilon$.[3]

Figure 3.4 shows the Parser component at parsing time when it is invoked by MALTPARSER function, using the function PARSE(x), to parse a sentence x. The Parser delegates the invocation to the parsing algorithm P_i using the function PARSE$_i(x)$, which parses the sentence according to the parsing algorithm P_i.

---

[3] The context stack v is only used in combination with Covington's algorithm.

33
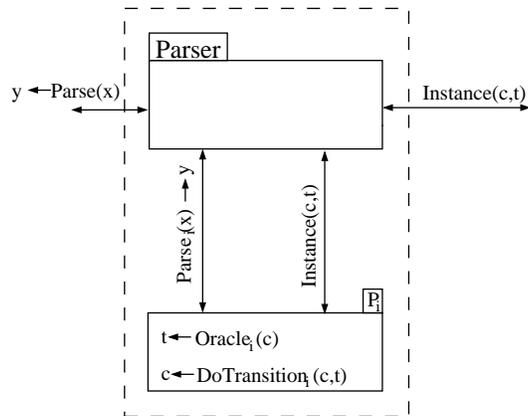
Figure 3.5: The parser compontent at learning time

When the parsing algorithm requests a transition it invokes the PRE-DICT(c) function. The Parser component forwards the request to the Guide component, which predicts the next transition t using the current parser configuration c. If the transition t is not permissible according to the current parser configuration c, instead a default transition is used. The transition t is applied using the DOTRANSITION$_i(c, t)$ function, which uses the transition system defined by the parsing algorithm P_i. Finally, the parsing algorithm P_i returns the analysis y via the Parser component to the MALTPARSER function.

At learning time, the procedure is similar (see Figure 3.5). The difference is that the parsing algorithm P_i uses the ORACLE$_i()$ function to derive the correct transition t from a gold standard treebank T_t and it sends the transition t together with the parser configuration c to the Guide component by using the function INSTANCE(c,t).

We will now continue to define the functions: PARSE$_i(x)$, ORACLE$_i(c)$, PERMISSIBLE$_i(c, t)$, DEFAULTTRANSITION$_i()$ and DOTRANSITION$_i(c, t)$, for each parsing algorithm currently implemented, i.e., Nivre's and Covington's algorithm.

Implementation of Nivre's algorithm The PARSE$_i(x)$ function for Nivre's algorithm is defined in section 2.3.1 together with the two transition systems (arc-eager and arc-standard) implemented in DOTRANSITION$_i(c, t)$. In addition, we need to define ORACLE$_i(c)$, PERMISSIBLE$_i(c, t)$ and

34

DEFAULTTRANSITION$_i$() for the two versions. ORACLE-NIVREARCEAGER
for the arc-eager version is defined by Nivre (2006):

> ORACLE-NIVREARCEAGER($c = (\sigma|i, j|\tau, \upsilon, h, d), h_g, d_g$)
> 1 **if** $h_g(i) = j$
> 2   **return** LEFT-ARC($d_g(i)$)
> 3 **else if** $h_g(j) = i$
> 4   **return** RIGHT-ARC($d_g(j)$)
> 5 **else if** $\exists k \in \sigma \ (h_g(j) = k$ **or** $h_g(k) = j)$
> 6   **return** REDUCE
> 7 **else**
> 8   **return** SHIFT

We need to distinguish between a head function h and a gold standard head
function h_g. The first head function h uses the head from the partially
built structure and the gold standard head function h_g takes the head from
a treebank. The same goes for the dependency function d, where the gold
standard dependency function d_g use the dependency type in the treebank.
The returned transition is the same as the transition defined by the transition
system for the arc-eager version. If there exists an arc between the token on
top of the stack i and the next input token j, according to the gold standard,
the correct transition is either LEFT-ARC or RIGHT-ARC. Both transitions
have a correct dependency type d_g according to the gold standard. If there is
no arc between those two nodes, the transition is either REDUCE or SHIFT. If
j is linked to a token to the left of i the REDUCE transition is used, otherwise
the SHIFT transition is used.

The function PERMISSIBLE-NIVREARCEAGER is used to determine if a
transition t is allowed or not:

> PERMISSIBLE-NIVREARCEAGER($c = (\sigma|i, j|\tau, \upsilon, h, d), t$)
> 1 **if** $t = $ LEFT-ARC **and** $d(i) \neq r_0$
> 2   **return** false
> 3 **else if** $t = $ **Reduce and** $d(i) = r_0$
> 4   **return** false
> 5 **else**
> 4   **return** true

The two following functions ORACLE-NIVREARCSTANDARD and PERMISSIBLE-
NIVREARCSTANDARD define the oracle and permissible functions for the arc-
standard version:

35

$\textsc{Oracle-NivreArcStandard}(c = (\sigma|i,j|\tau,\upsilon,h,d), h_g, d_g)$
1 **if** $h_g(\sigma[i]) = \tau[j]$
2   **return** $\textsc{Left-Arc}(d_g(\sigma[i]))$
3 **else if** $h_g(\tau[j]) = \sigma[i]$
4   **return** $\textsc{Right-Arc}(d_g(\tau[i]))$
5 **else**
6   **return** $\textsc{Shift}$

$\textsc{Permissible-NivreArcStandard}(c = (\sigma|i,j|\tau,\upsilon,h,d), t)$
1 **if** $t = \textbf{Left-Arc}$ **and** $d(i) \neq r_0$
2   **return** false
3 **else**
4   **return** true

The default transition for both versions is the $\textsc{Shift}$ transition, i.e., if the transition proposed by the guide during parsing is not permissible, then the $\textsc{Shift}$ is used instead.

Implementation of Covington's algorithm   The $\textsc{Parse}_i(x)$ function for Covington's algorithm is defined in section 2.3.1 together with the transition system (projective), implemented in $\textsc{DoTransition}_i(c, t)$. This thesis will only describe the **Projective** graph condition, because we only use this version in the experiments. The $\textsc{Oracle-Covington}$ function derives the correct transition in the following way:

$\textsc{Oracle-Covington}(c = (\sigma|i,j|\tau,\upsilon,h,d), h_g, d_g)$
1 **if** $h_g(i) = j$
2   **return** $\textsc{Left-Arc}(d_g(i))$
3 **else if** $h_g(j) = i$
4   **return** $\textsc{Right-Arc}(d_g(j))$
5 **else**
6   **return**$\textsc{Reduce}$

Covington's oracle function only derives three transitions because the $\textsc{Shift}$ transition is performed implicitly by the algorithm. The $\textsc{Reduce}$ transition is the default transition of Covington's algorithm, since there is no **Shift** transition. Permissibility is checked by:

$\textsc{Permissible-Covington}(c = (\sigma|i,j|\tau,\upsilon,h,d), t)$
1 **if** $t = \textbf{Left-Arc}$ **and** $d(i) \neq r_0$
2   **return** false
3 **else if** $t = \textbf{Right-Arc}$ **and** $d(j) \neq r_0$
4   **return** false
5 **else**
6   **return** true

```
<fspec>  ::= <feat>+
<feat>   ::= <lfeat>|<nlfeat>
<lfeat>  ::= LEX\t<dstruc>\t<off>\t<suff>\n
<nlfeat> ::= (POS|DEP)\t<dstruc>\t<off>\n
<dstruc> ::= (STACK|INPUT|CONTEXT)
<off>    ::= <nnint>\t<int>\t<nnint>\t<int>\t<int>
<suff>   ::= <nnint>
<int>    ::= (...|-2|-1|0|1|2|...)
<nnint>  ::= (0|1|2|...)
```

Figure 3.6: The syntax for defining a feature model in an external feature specification.

The Parser component interacts with the Guide component by using the functions INSTANCE and PREDICT, and the next section 3.2.4 describes how this interaction is performed from the Guide's point of view.

3.2.4 Guide

The main tasks of the Guide component is to induce a transition by using a classifier during parsing (the PREDICT function) and to provide a learner with training instances during learning (the INSTANCE function). Moreover, the Guide is also responsible for extracting the feature values that correspond to the specified feature model.

The MaltParser system comes with a formal specification language for feature functions, which enables the user to define arbitrarily complex feature models in terms of address functions, attribute functions and mapping functions (see section 3.1.2). Each feature model is defined in a *feature specification* file, which allows users to define feature models without rebuilding the system.

The feature specification uses the syntax described in Figure 3.6. Each feature is specified on a single line, consisting of at least two tab-separated columns. Below follows a description of each column:

1. The first column defines the feature type to be part-of-speech (POS), dependency (DEP) or word form (LEX), corresponding to the attribute functions p, d and w in section 3.1.[4]

---

[4]With the CoNLL-X shared task format it is possible to define three additional feature types: lemma (LEMMA), part-of-speech of a coarse-grained tagset (CPOS), syntactic and/or morphological features (FEATS).

37

2. The second column identifies one of the main data structures in the parser configuration: `STACK` (corresponding to σ), `INPUT` (corresponding to τ) or `CONTEXT` (corresponding to υ). The third alternative, `CONTEXT`, is relevant only together with Covington's algorithm in non-projective mode.

3. The third column defines a list offset i which can only be positive and which identifies the $i+1$th token in the list/stack specified in the second column (i.e. $\sigma[i]$, $\tau[i]$ or $\upsilon[i]$).

4. The fourth column defines a linear offset i, which can be positive (forward/right) or negative (backward/left) and which refers to (relative) token positions in the original input string.

5. The fifth column defines an offset i in terms of the function h (head), which has to be non-negative and which specifies i applications of the h function to the token identified through preceding offsets.

6. The sixth column defines an offset i in terms of the functions lc (the leftmost child) or rc (the rightmost child), which can be negative ($|i|$ applications of lc), positive (i applications of rc), or zero (no applications).

7. The seventh column defines an offset i in terms of the functions ls (the next left sibling) or rs (the next right sibling), which can be negative ($|i|$ applications of ls), positive (i applications of rs), or zero (no applications).

8. If the first column specifies the attribute function w (`LEX`), the eighth column defines a mapping function which specifies a suffix of length n of the word form w. By convention, if $n = 0$, the entire word form is included; otherwise only the n last characters are included in the feature value.

Let us consider an example:

```
POS     STACK   0       0       0       0       0
POS     INPUT   0       0       0       0       0
POS     INPUT   1       0       0       0       0
DEP     STACK   0       0       0       0       0
LEX     INPUT   1       0       0       0       0       0
```

The first three feature are part-of-speech features; the first is the topmost token on the stack, $p(\sigma[0])$, and the other two are $p(\tau[0])$ and $p(\tau[1])$. The feature defined on the fourth line is the dependency type of the token located

38
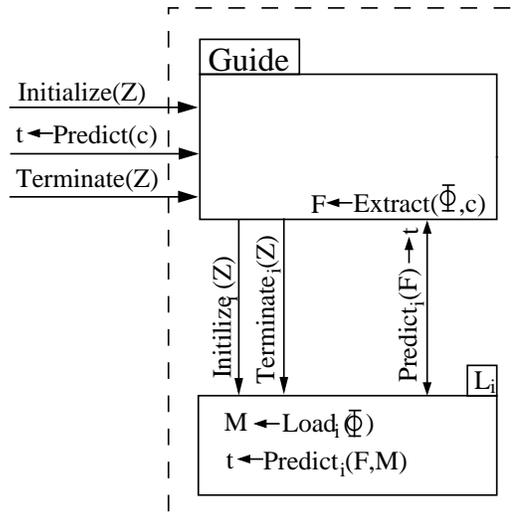
Figure 3.7: The Guide component at parsing time

at the top of the stack, $d(\sigma[0])$. Finally, the last feature is the word form of the token immediately after the next input token, i.e., $w(\tau[1])$. As syntactic sugar, any feature definition can be truncated if all remaining integer values are zero. The same example with this truncation:

```
POS     STACK
POS     INPUT
POS     INPUT   1
DEP     STACK
LEX     INPUT   1
```

Figure 3.7 illustrates how the Guide component is used during parsing. The Guide is initialized by the invocation of the function INITIALIZE(Z), which loads the feature specification. The Guide delegates invocation to the learner interface L_i, a subcomponent of the Guide, which loads the model M by invoking the LOAD$_i(\Phi)$ function. The Parser component invokes the Guide when it needs a prediction by invoking the function PREDICT(c), where c is the current parser configuration. The Guide extracts the feature vector $\Phi(H_i, x) = (v_1, \ldots, v_p)$ according to the feature specification by using the function Extract(Φ, c). It stores the feature values $v_1, \ldots, v_p$ in a dedicated data structure F. The learner interface L_i is invoked by the function

39
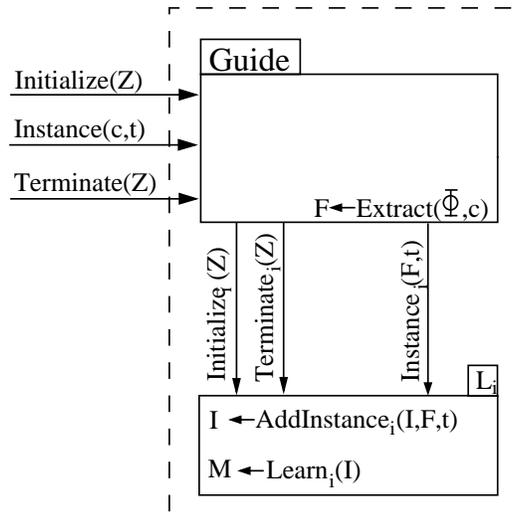
Figure 3.8: The Guide component at learning time

$\textsc{Predict}_i(F)$, which iterates through the data structure F in a linear fashion and formats the feature values according to the format required by the classifier. It proceeds with a call to the classifier, which predicts the transition given the model M and the feature values.

During learning the Guide is initialized with a feature specification in the same way as during parsing. The Guide is invoked by the Parser component using the function $\textsc{Instance}(c,t)$, which makes the Guide extract the feature values in the same way as during parsing, see Figure 3.7. It delegates the invocation to the learner interface L_i using the function $\textsc{Instance}_i(F,t)$, which makes the learner interface format the feature values and the transition in the required format for a particular learner. It proceeds by adding the feature values together with the transition to the sequence of training instances I.

When all the sentences in the gold standard treebank T_t have been parsed using the oracle function, the system will terminate all its components. During this event, the learner subcomponent L_i uses the sequence of training instances I to induce a model M.

In the current implementation each learner L_i is actually an interface to a machine learning package. The interface prepares the set of training instances, provided by the Guide, for the specific package and invokes the

40

appropriate functions to learn a model or predict a transition. MaltParser comes with two different learning algorithms:

- TiMBL (Tilburg Memory-Based Learner) is a software package for memory-based learning and classification (Daelemans and Van den Bosch 2005), which directly handles multi-valued symbolic features.

- LIBSVM library (Chang and Lin 2001) is a software library for SVM classification, which is able to handle multi-class classification.

Both TiMBL and LIBSVM have a wide variety of parameters, which can be tuned for a specific learning task.

TiMBL *interface* The TiMBL interface interacts with the TiMBL learner by invoking functions specified in an API (Application Programming Interface). MBL is briefly described in section 2.3.3. The idea during learning is to collect the values of different features from the training data together with the correct class according to the training data. At classification time it reuses these instances by finding similarities and in that way tries to find the most appropriate class. All TiMBL specific parameters can be used together with MaltParser, which makes it possible to optimize the model.

Using the Swedish sentence shown in Figure 2.1 as training data for Nivre's arc-eager algorithm in section 2.3.1 with the example feature model explained in section 3.2.4, the TiMBL interface will collect the following training instances:

```
nn.nom vb.fin ab ROOT också LA_SUB
vb.fin ab pp ROOT för RA_ADV
ab pp nn.nom ADV mopedister RE
vb.fin pp nn.nom ROOT . RA_ADV
pp nn.nom mad ADV #null RA_PR
nn.nom mad #null PR #null RE
pp mad #null ADV #null RE
vb.fin mad #null ROOT #null RA_IP
```

The first three items are the values of the part-of-speech features $p(\sigma[0])$, $p(\tau[0])$ and $p(\tau[1])$, the fourth item is the value of the dependency type feature $d(\sigma[0])$, and the fifth item is the value of the lexical feature $w(\tau[1])$. A null-value is used when there is no value, for example, if the feature $p(\tau[1])$ points beyond the last input token of the sentence. The last item is the transition and if it is LEFT-ARC (LA) or RIGHT-ARC (RA) the dependency type is supplied after the underscore sign. Only the transitions marked with

41

N (nondeterministic) in the example in section 2.3.1 are used as training instances.

When the parser (in learning mode) has processed all the training data, the learner interface will invoke the TiMBL learner, which creates a model of all the training data. In parsing mode, the learner interface will invoke the TiMBL learner with the values of the same features. For example, if we have the same parser configuration as for the first training instance the learner interface will invoke the TiMBL classifier with `nn.nom vb.fin ab ROOT också` and hopefully the TiMBL learner will predict the parse action `LA_SUB`.

LIBSVM *interface* We will now continue with describing the LIBSVM interface. SVM learning relies on kernel functions to induce a maximum-margin hyperplane classifier at learning time, which can be used to predict the next transition at parsing time. A more detailed description of SVM can be found in section 2.3.3.

The process of collecting training instances is similar to the interface for the TiMBL learner, but all feature values and transitions are represented by a numerical value. This entails that all word types must be represented as numerical values and therefore a lexicon with all word types with their corresponding values is created before any learning takes place. In parsing mode, the lexicon is read before any parsing is done.

The LIBSVM library comes with many parameters, but only the parameters used in the experiments are explained. The experiments use the polynomial kernel $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d, \gamma > 0$, where d, γ and r are kernel parameters. Other parameters that are varied in experiments are the penalty parameter C, which defines the tradeoff between training error and the magnitude of the margin, and the termination criterion ϵ, which determines the tolerance of training errors.

In order to reduce the training time it is possible to divide the training data into smaller sets, according to a feature ϕ_s in the feature model, example $p(\tau[0])$, and train one set of classifiers for each smaller set. Similar techniques have previously been used by Yamada and Matsumoto (2003), among others, without significant loss of accuracy. In order to avoid too small training sets, it is possible to pool together all instances that have a frequency below a certain threshold t into a single set. These things are done by the interface, not by the LIBSVM learner. To handle these things, the interface is equipped with three parameters, the first parameter S handles the splitting strategies:

- **No splitting** (ns): There is no division at all, the parser will create a single model.

- **Head splitting** (hs): The training instances are divided into two sets

42

according to whether the topmost token of the stack has a head or not.

- **Feature splitting** (fs): The training instances are divided into smaller sets based on a splitting feature ϕ_s, which pool together all instances that share the same feature value. The number of training sets are at most $m + 1$, where m is the number of distinct feature values for the splitting feature ϕ_s. There will also be a special training set that pools together all low frequently values.

- **Combined splitting** (cs): This strategy combines the head and feature splitting strategy and there will be at most $2(m + 1)$ data sets.

The second parameter F determines which feature ϕ_s will be used for the splitting. The feature must be specified in the feature model and it must also be a part-of-speech or dependency type feature, i.e., splitting on lexical features is not allowed. The order in the feature model is important when this parameter used. An integer value is used to point out the feature for a particular feature type (starting from 0). There are two feature types, `P` corresponds to `POS` in the feature specification language and `D = DEP`. For example, if we want to use the second feature $p(\tau[0])$ or `POS INPUT 1` in the example feature model used above, we specify `P1; P` because it is a part-of-speech feature and 1 because it is the second feature of that type (indexing starting at 0).

Finally, there is a parameter T specifying the frequency threshold t that determines if a certain feature value should be pooled together with other values that occur less than t times in the training data.

In order to further reduce training time, we use *Light Weight Processes* or threads, to train SVM models in parallel when the training instances are divided into several sets.

This architecture and its implementation in the MaltParser system is validated with three experiments in the chapter 4. We will see if MaltParser can produce labeled dependency structures with high accuracy for several languages. We will also investigate if the system is flexible, i.e., if it is possible to vary parsing algorithm, feature model and learner without rebuilding the system. The LIBSVM interface is also explored with several experiments to investigate, for instance, the splitting strategies.

Chapter 4

Experiments

This chapter contains a presentation of experiments which have already been introduced in section 1.1 in relation to validation questions:

1. **Validation of the implementation**: Does MaltParser realize the underlying architecture, so that it is possible to vary parsing algorithm, feature model and learning method independently? We will investigate this by varying three parsing algorithms – Nivre's arc-eager, Nivre's arc-standard and Covington's projective, two feature models – one unlexicalized and one lexicalized, and two learning methods – MBL and SVM. All these combinations will be applied to three languages – Swedish, English and Chinese.

2. **Investigation of the SVM interface**: How do the special properties of the SVM interface affect parsing accuracy and time efficiency? How can learning and parsing efficiency be improved without sacrificing accuracy? One of the disadvantages of SVM is that it takes a lot of time to train a model and also to use this model for prediction. The time efficiency degrades significantly with more training instances. MaltParser is equipped with mechanisms for dividing the training data into smaller sets and training separate classifier for each set. These experiments will explore the different strategies for dividing the data and which features should be used.

3. **Comparison of MBL and SVM**: Which of the learning methods is best suited for the task of inductive labeled dependency parsing, taking both parsing accuracy and time efficiency into account? This study compares MBL and SVM to investigate if there is any difference regarding parsing accuracy and time efficiency. The study is again based on three languages – Chinese, English, and Swedish – and on five different feature models of varying complexity, with a separate optimization of learning algorithm parameters for each combination of language and feature model.

The chapter is structured as follows. Section 4.1 presents the data sets for the three languages – Swedish, English and Chinese. Section 4.2 discusses

45

evaluation metrics used in the experiments. The feature models used in experiments are listed in Section 4.3. Sections 4.4–4.6 present the results for the three validation questions stated in Section 1.1 and each section is concluded with a discussion.

4.1 Data Sets

4.1.1 Swedish

Sweden was one of the pioneers in extending corpora with syntactic annotation. Talbanken in the 70s (Einarsson 1976a; Einarsson 1976b) and SynTag in the 80s (Järborg 1986) were both based on Swedish data. Unfortunately, there are no activities for creating a large-scale treebank for Swedish in a modern format. In the absence of Swedish treebanks, we have converted Talbanken into a more convenient format for parsers and treebank tools, which has resulted in three versions (Talbanken76, Talbanken03 and Talbanken05).

The original annotated corpus, which we call Talbanken76, is a manually annotated corpus of both written and spoken Swedish, created at Lund University in the 1970s. The corpus contains close to 320,000 words, and it is divided into four parts:

- "Professionell prosa" (Professional prose) consists of material taken from textbooks, newspapers and information brochures and comprises about 85,000 words.
- "Gymnasistsvenska" (Swedish by senior high school students) contains essays written by senior high school students on the topic "Familjen och äktenskapet än en gång" (Family and marriage once again) and comprises about 85,000 words.
- "Samtal och debatt" (Conversation and debate) consists of transcriptions of conversations and debates on different topics and has about 75,000 words.
- "Boråsintervjuerna" (The Borås interviews) comprises interviews with different persons on the topic of immigrants and has about 75,000 words.

The original annotation scheme of Talbanken76 follows the guidelines of the MAMBA scheme (Teleman 1974), which consists of two layers. The lexical analysis layer, comprising part-of-speech information and morphological features, and the syntactic analysis layer, consisting of grammatical functions.

The goal of the first conversion (Talbanken03) was to convert, with very high accuracy, the annotated sentences to projective dependency graphs.

46

ADV	Adverbial modifier
APP	Apposition
ATT	Attribute (adnominal modifier)
CC	Coordination (conjunction or second conjunct)
DET	Determiner
ID	Non-first element of multi-word expression (idiom)
IM	Infinitive dependent on infinitive marker
INF	Infinitival complement
IP	Punctuation
OBJ	Object
PR	Complement of preposition
PRD	Predicative complement
ROOT	Dependent of special root node
SUB	Subject
UK	Head verb of subordinate clause dependent on complementizer
VC	Verb chain (nonfinite verb dependent on other verb)
XX	Unclassifiable dependent

Table 4.1: Dependency types in Swedish treebank

Only the professional prose part was included in the conversion process and also the original fine-grained classification of grammatical functions was reduced into a set of 17 dependency types, which are listed in Figure 4.1.

We have retagged the converted treebank with an HMM-based part-of-speech tagger with suffix smoothing (Hall 2003). The part-of-speech tagger was trained on the Stockholm Umeå Corpus (Ejerhed and Källgren 1997), with a tagset containing 150 tags, and has an estimated accuracy of 95.1%. We have used a pseudo-randomized method, dividing the data into 10 sections of equal size, where sentence i is allocated to section i mod 10. We have used sections 1–9 for 9-fold cross-validation during development and section 0 for final evaluation. Different characteristics of data splits are presented in table 4.2.

In the second conversion (Talbanken05), all four parts of the original corpus were converted into two phrase structure versions encoded in TIGER-XML and one dependency structure version encoded in Malt-XML (Nilsson et al. 2005; Nivre et al. 2006). The work on these conversions was parallel to the work on this thesis and therefore all presented results are obtained using Talbanken03.

47

Table 4.2: **The professional prose section of Talbanken** is divided into 10 sections. Section 0 is saved for the final evaluation. Sections 1-9 are used for 9-fold cross validation. #T = Number of tokens, #S = Number of sentences, T/S = Average sentence length and #W = Number of distinct word forms.

Section	#T	#S	T/S	#W
0	9841	631	15.60	3501
1–9	86757	5685	15.26	15277
Total	**97598**	**6316**	**15.45**	**16345**

4.1.2 English

We use the Wall Street Journal section of the Penn Treebank (Marcus et al. 1993) for the English experiments. The treebank uses the Penn Treebank II annotation scheme (Bies et al. 1995), which consists of constituency analysis and some functional annotation.

The constituent structure has been converted to dependency structure using the head percolation table of Yamada and Matsumoto (2003), which transforms the treebank into an unlabeled dependency structure. Figure 4.1 shows the head percolation table presented in Nivre (2006). Every constituent label in the Penn Treebank is listed in the first column, and for each label the search direction is presented in the second column (from the right (R) or from the left (L)). A priority list of potential head child categories is given in the third column; the vertical bar | means that the categories have the same priority. For example, if the constituent label is NP, the search starts from the right for a child with POS, NN, NNP, NNPS or NNS. If we find a child with one of these types, then this child is chosen as the head; otherwise the search proceeds with NX, etc. If the search fails to find a child with a label in the ordered list, then the rightmost child is chosen as the head because of the direction of the search.

We want to perform labeled dependency parsing and therefore we need to infer the labels for each arc in the unlabeled dependency structure. This task is not a trivial problem, but Nivre (2006) has used 11 rules based on Collins (1999) rules. We can use the original phrase labels and the parts-of-speech on the lexical level to derive an arc label r for an arc $i \rightarrow j$ from a local constituent, where w_i is the head word h of the head child and w_j is the head word d of a non-head child. Let M, H and D be the original labels on the mother node, head child and child, respectively, except that H is TAG if the head child is labeled with a part-of-speech tag. Figure 4.2 shows the

48

NP	R	POS|NN|NNP|NNPS|NNS NX JJR CD JJ JJS RB QP NP
ADJP	R	NNS QP NN $ ADVP JJ VBN VBG ADJP JJR NP JJS DT
		FW RBR RBS SBAR RB
ADVP	L	RB RBR RBS FW ADVP TO CD JJR JJ IN NP JJS NN
CONJP	L	CC RB IN
FRAG	L	
INTJ	R	
LST	L	LS :
NAC	R	NN|NNS|NNP|NNPS NP NAC EX $ CD QP PRP VBG JJ
		JJS JJR ADJP FW
PP	L	IN TO VBG VBN RP FW
PRN	R	
PRT	L	RP
QP	R	$ IN NNS NN JJ RB DT CD NCD QP JJR JJS
RRC	L	VP NP ADVP ADJP PP
S	R	TO IN VP S SBAR ADJP UCP NP
SBAR	R	WHNP WHPP WHADVP WHADJP IN DT S SQ SINV
		SBAR FRAG
SBARQ	R	SQ S SINV SBARQ FRAG
SINV	R	VBZ VBD VBP VB MD VP S SINV ADJP NP
SQ	R	VBZ VBD VBP VB MD VP SQ
UCP	L	
VP	L	VBD VBN MD VBZ VB VBG VBP VP ADJP NN NNS NP
WHADJP	R	CC WRB JJ ADJP
WHADVP	L	CC WRB
WHNP	R	WDT WP WP$ WHADJP WHPP WHNP
WHPP	L	IN TO FW
NX	R	POS|NN|NNP|NNPS|NNS NX JJR CD JJ JJS RB QP NP
X	R	

Figure 4.1: Head percolation table for the Penn Treebank

49
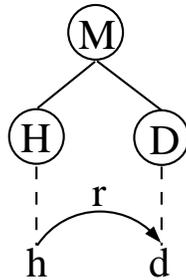
Figure 4.2: The setup of the local constituent for deriving the arc label r.

setup of the local constituent for deriving the arc label r, where H and D are recursively connected to the words h and d. Nivre (2006) has formulated a set of rules for choosing the arc label r, $i \xrightarrow{r} j$. In order of descending priority, the rules are as follows:

1. If D is a punctuation category, $r =$ P.
2. If D contains the function tag SBJ, $r =$ SBJ.
3. If D contains the function tag PRD, $r =$ PRD.
4. If $M =$ VP, $H =$ TAG and $D =$ NP (without any function tag), $r =$ OBJ.
5. If $M =$ VP, $H =$ TAG and $D =$ VP, $r =$ VC.
6. If $M =$ SBAR and $D =$ S, $r =$ SBAR.
7. If $M =$ VP, S, SQ, SINV or SBAR, $r =$ VMOD.
8. If $M =$ NP, NAC, NX or WHNP, $r =$ NMOD.
9. If $M =$ ADJP, ADVP, QP, WHADJP or WHADVP, $r =$ AMOD.
10. If $M =$ PP or WHPP, $r =$ PMOD.
11. Otherwise, $r =$ DEP.

For example, if there is a local constituent with a mother node $M =$ VP, which has a child $D =$ NP without any function tag and a head child H with an arbitrary part-of-speech tag (TAG), then we can derive an arc label $r =$ OBJ for the arc $i \xrightarrow{r} j$. The set of dependency types R is listed in Figure 4.3, which also includes the special root label $r_0 =$ ROOT.

Table 4.4 shows the characteristics for the different splits of data. Sections 2–21 are used for training, section 0 for development testing and section 23 for final evaluation. Sections 0 and 23 are part-of-speech tagged using the same part-of-speech tagger as for Swedish, but trained on section 2–21, and has an estimated accuracy of 96.6% for English.

50

AMOD	Modifier of adjective or adverb (phrase adverbial)
DEP	Other dependent (default label)
NMOD	Modifier of noun (including complement)
OBJ	Object
P	Punctuation
PMOD	Modifier of preposition (including complement)
PRD	Predicative complement
ROOT	Dependent of special root node
SBAR	Head verb of subordinate clause dependent on complementizer
SUB	Subject
VC	Verb chain (nonfinite verb dependent on other verb)
VMOD	Modifier of verb (sentence or verb phrase adverbial)

Table 4.3: Inferred dependency types for Penn Treebank both for English and Chinese

Table 4.4: **The Wall Street Journal section of the Penn Treebank** is divided into three parts (training, devtest and etest). #T = Number of tokens, #S = Number of sentences, T/S = Average sentence length and #W = Number of distinct word forms.

Section	#T	#S	T/S	#W
training (2–21)	950028	39832	23.85	44389
devtest (0)	46451	1921	24.18	7903
etest (23)	56684	2416	23.46	8421
Total	**1053163**	**44169**	**23.84**	**46759**

51

4.1.3 Chinese

The Chinese data are taken from the Penn Chinese Treebank (CTB), version
5.1 (Xue et al. 2002; Xue et al. 2004), and the texts are mostly from Xinhua
newswire, Sinorama news magazine and Hong Kong News. The annotation
of CTB is based on a combination of constituent structure and grammatical
functions such as *subject*, *object*, etc.

CTB has been converted into dependency structure in the same way as the
data for English, with a head percolation table created by a native speaker
for the purpose of machine translation. Table 4.5 shows the head percolation
table used for transforming CTB into dependency structure. The format is
slightly different from the one used for English. The first column specifies
all the constituent labels in the CTB. For each label there are ordered lists
of potential head child categories, separated by semi-colons. The lists are
shown in the second column and each list begins with a direction of search
(from the right to left (r) or from the left to right (l)) and continues with
potential head child categories. If the search cannot find a head in the lists,
the third column is used to determine if the rightmost child (r) or leftmost
child (l) should be used as a head. For example, for the constituent label
ADJP, we start searching from the right for a child of type ADJP or JJ. The
first child matching this condition is chosen as the head. If there is no match,
we continue with the next list. If a child of type AD, NN or CS cannot be
found, we use the rightmost child as the head.

The arc labels are derived in the same way as for English, except that
another set of rules is used. We also use the same set of dependency types
R listed in Figure 4.3. In order of descending priority, the Chinese labeling
rules are as follows:

1. If D is a punctuation category, $r = $ P.
2. If D contains the function tag SBJ, $r = $ SBJ.
3. If D contains the function tag PRD, $r = $ PRD.
4. If $M = $ VP, $H = $ TAG and $D = $ NP-OBJ, $r = $ OBJ.
5. If $M = $ VP, $H = $ TAG and $D = $ VP, $r = $ VC.
6. If $M = $ VCD, VCP, VRD or VSB, $r = $ VC.
7. If $M = $ VNV or VPT, $H = $ TAG, $r = $ VC.
8. If $M = $ CP and $D = $ IP, $r = $ SBAR.
9. If $M = $ VP, IP, SQ, SINV or CPQ, $r = $ VMOD.
10. If $M = $ NP, NAC, NX or WHNP, $r = $ NMOD.
11. If $M = $ ADJP, ADVP, QP, WHADJP or WHADVP, $r = $ AMOD.
12. If $M = $ PP or WHPP, $r = $ PMOD.

52

ADJP	r ADJP JJ; r AD NN CS	r
ADVP	r ADVP AD	r
CLP	r CLP M	r
CP	r DEC SP; l ADVP CS; r CP IP	r
DNP	r DNP DEG; r DEC	r
DP	l DP DT	l
DVP	r DVP DEV	r
FRAG	r VV NR NN	r
INTJ	r INTJ IJ	r
IP	r IP VP; r VV	r
LCP	r LCP LC	r
LST	l LST CD OD	l
NP	r NP NN NT NR QP	r
PP	l PP P	l
PRN	r NP IP VP NT NR NN	r
QP	r QP CLP CD OD	r
UCP		r
VCD	r VCD VV VA VC VE	r
VCP	r VCP VV VA VC VE	r
VNV	r VNV VV VA VC VE	r
VP	l VP VA VC VE VV BA LB VCD VSB VRD VNV VCP	l
VPT	r VNV VV VA VC VE	r
VRD	r VRD VV VA VC VE	r
VSB	r VSB VV VA VC VE	r
WHNP	r WHNP NP NN NT NR QP	r
WHPP	l WHPP PP P	l

Table 4.5: Head percolation table for the Penn Chinese Treebank

Table 4.6: **The Penn Chinese Treebank** is divided into three parts (training, devtest and etest). #T = Number of tokens, #S = Number of sentences, T/S = Average sentence length and #W = Number of distinct word forms.

Section	#T	#S	T/S	#W
training (1-8)	408740	15044	27.17	33556
devtest (9)	50243	1880	26.73	10404
etest (0)	49781	1880	26.48	10301
Total	**508764**	**18804**	**27.06**	**37449**

53

13. Otherwise, $r = \text{DEP}$.

One often underestimated parameter in the evaluation of parsing accuracy is the division of data into training, development and evaluation set. Levy and Manning (2003) report differences of up to 10 percentage points in parsing accuracy for different splits of CTB 2.0. All files in CTB 5.1 were merged into a large data set by concatenating the files in lexical order. We have used the same kind of pseudo-randomized data split as for Swedish, and the treebank is divided into three sections (*training*, *devtest* and *etest*). Characteristics are presented in table 4.6. The section *etest* is saved for the final evaluation; *training* is used for training the parser with section *devtest* used for tuning the settings and parameters.

Written Chinese does not have a natural delimiter between words, which means that the text needs a more complex automatic word-segmenter, compared to, for instance, English. Part-of-speech tagging is also harder than for standard European languages. Therefore, most parsing results for Chinese reported in the literature are based on gold standard segmentation and tagging. This is true also for the results presented in this thesis.

4.2   Evaluation Metrics

In Section 2.1 we defined four requirements: *robustness*, *disambiguation*, *accuracy* and *efficiency*. The first two requirements are built into the system, since the parser will always produce exactly one analysis. The data structures are limited to handling only one analysis and in the worst case scenario, the parser will create a graph that has all tokens attached to the special root node. In practice it is impossible to fulfil the *accuracy* requirement. Instead we will use this to evaluate the system. It is interesting to see how often the parser parses a sentence completely correctly (*exact match*), but this measure is quite hard. It is therefore common to measure the accuracy based on how often a token is correctly parsed (*attachment score*). We will use four accuracy metrics defined as follows:

1. **Unlabeled attachment score** (AS_U) is the proportion of tokens that are assigned the correct head (regardless of dependency type).

2. **Labeled attachment score** (AS_L) is the proportion of tokens that are assigned the correct head and the correct dependency type.

3. **Unlabeled exact match** (EM_U) is the proportion of sentences that are assigned a completely correct dependency graph without considering dependency type labels.

54

Table 4.7: Five feature models used in the experiments

Feature	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5
$p(\sigma_0)$	+	+	+	+	+
$p(\tau_0)$	+	+	+	+	+
$p(\tau_1)$	+	+	+	+	+
$p(\tau_2)$				+	+
$p(\tau_3)$				+	+
$p(\sigma_1)$					+
$d(\sigma_0)$		+	+	+	+
$d(lc(\sigma_0))$		+	+	+	+
$d(rc(\sigma_0))$		+	+	+	+
$d(lc(\tau_0))$		+	+	+	+
$w(\sigma_0)$			+	+	+
$w(\tau_0)$			+	+	+
$w(\tau_1)$					+
$w(h(\sigma_0))$					+

4. **Labeled exact match** (EM_L) is the proportion of sentences that are assigned a completely correct dependency graph, which also takes dependency type labels into account.

Another common procedure is that attachment scores are presented as mean scores per token, and punctuation tokens are excluded from all counts. For the third experiment in section 4.6 we have performed a McNemar test of significance at $\alpha = 0.01$.

The last requirement *efficiency* is fulfilled as far as the parsing algorithms concerned; all algorithms used have a polynomial complexity of at most degree 2. The complexity does not say anything about the classification task, which is considered as a constant factor. Therefore we will for some experiments measure the time it takes to learn a model or parse unseen data:

1. **Learning time** (LT) is the time it takes to train a model on hardware system H given a training sample T_t, parsing algorithm P_i, feature model Φ, learner L_i and learner parameter settings L_p.

2. **Parsing time** (PT) is the time it takes to parse an evaluation sample T_e on hardware system H using parsing algorithm P_i, feature model Φ, learner L_i and learner parameter settings L_p.

All experiments are performed on an AMD 64-bit processor running Linux.

Table 4.8: The baseline parsing accuracy

Language	Left		Right	
	$\mathbf{AS}_U$	$\mathbf{AS}_L$	$\mathbf{AS}_U$	$\mathbf{AS}_L$
Swedish	21.5	3.4	35.8	5.2
English	29.3	19.8	19.7	5.9
Chinese	38.3	17.4	7.9	0.3

4.3   Feature Models

Table 4.7 describes the five feature models Φ_1–Φ_5 used in the experiments, with features specified in column 1 using the functional notation defined in Section 3.1.2. Thus, $p(\sigma_0)$ refers to the part-of-speech of the top token, while $d(lc(\tau_0))$ picks out the dependency type of the leftmost child of the next token. It is worth noting that models Φ_1–Φ_2 are unlexicalized, since they do not contain any features of the form $w(\alpha)$, while models Φ_3–Φ_5 are all lexicalized to different degrees. Nivre (2006) has performed extensive feature selection experiments for MBL and with data from Swedish and English, and the best feature model in these experiments was the feature model Φ_5. These features have also been used in several experiments before and have given reasonable accuracy for several languages (Nivre and Hall 2005).

4.4   Experiment I: Validation

The implementation of the MaltParser system, described in section 3.2, has to be validated in some way. The system should produce well-formed dependency graphs with fairly high accuracy, but how do we define success for this task? One way of defining success is to compare the results with a baseline. We could use a randomized transition sequence as baseline and then measure the accuracy, but a randomized baseline would probably perform very badly. We will therefore use a baseline that uses left or right branching, i.e., that links each node to its successor or predecessor in the input string. Table 4.8 shows the accuracy for this baseline for the three languages both with left and right branching.

Another way to validate the performance of the system is to check whether the output complies with the conditions of well-formedness. For example, all sentences in the output should be assigned projective dependency graphs.

Flexibility is another requirement on the implementation. It should be possible to vary learning method, feature model and parsing algorithm inde-

56

pendently of each other without rebuilding the system.

4.4.1 Experimental Setup

This experiment checks whether the MaltParser system meets the requirements of the architecture and whether it is possible to vary the learning method, the feature model and the parsing algorithm independently. We use the following setup:

- Data from three different languages – Chinese, English, and Swedish
- Two learning methods – MBL and SVM
- Two feature models – Φ_2 and Φ_5 (see table 4.7)
- Three parsing algorithms – Nivre's arc-eager, Nivre's arc-standard and Covington's projective

The parameter settings for both learners are kept constant for all runs in this experiment. We use the same settings for the TiMBL learner that Nivre (2006) used in the final evaluation for Swedish, where the number of nearest neighbors is set to $k = 5$ and the Modified Value Difference Metric (MVDM) is used to compute distances between feature values. Inverse distance-weighted class voting (ID) is used to determine the majority class. MVDM is used down to $l = 3$; below that threshold the simple Overlap metric is used.

The parameter settings of the LIBSVM learner are also kept constant throughout all experiments. We use a polynomial kernel of degree 2 with the kernel parameters $\gamma = 0.2$ and $r = 0$. The default settings are used for the penalty parameter $C = 1$. The termination criterion e is assigned the highest possible value 1, which usually decreases the learning time dramatically compared to the default setting 0.001. The training instances for the SVM learner are divided based on the feature $p(\tau_0)$.

The experiments for Swedish are conducted using nine-fold cross-validation and for Chinese and English the development test set is used.[1]

4.4.2 Results and Discussion

Table 4.9 shows the parsing accuracy for all runs in this experiment. Only the attachment scores (both unlabeled and labeled) are presented in the table, because we are mainly interested in validating that the implementation meets the requirements of the architecture.

If we compare the baseline with the results in table 4.9, we can see that the attachment scores are significantly better for all feature models and all

---

[1] The final evaluation sets and significance tests are saved for the experiment described in Section 4.6.

57

Table 4.9: Parsing accuracy for three parsing algorithms, two feature models (FM), two learning methods (LM). The accuracy is measured by the attachment score (AS) metrics; U: unlabeled, L: labeled.

FM	Algorithm	LM	Swedish		English		Chinese	
			$\mathbf{AS}_U$	$\mathbf{AS}_L$	$\mathbf{AS}_U$	$\mathbf{AS}_L$	$\mathbf{AS}_U$	$\mathbf{AS}_L$
Φ_2	Arc eager	MBL	81.9	74.8	81.5	78.5	72.9	70.5
		SVM	**82.3**	**75.5**	**81.7**	**78.7**	73.2	70.9
	Arc standard	MBL	77.9	70.5	79.6	76.7	73.3	71.0
		SVM	77.8	70.9	79.6	76.7	**73.6**	**71.4**
	Covington	MBL	80.5	73.5	80.9	78.0	73.0	70.6
		SVM	80.8	74.3	81.2	78.4	73.3	71.0
Φ_5	Arc eager	MBL	85.8	81.5	87.6	85.8	80.4	78.6
		SVM	**86.4**	**82.6**	**88.9**	**87.4**	83.9	82.4
	Arc standard	MBL	83.2	78.7	87.5	85.7	81.6	79.8
		SVM	84.6	80.5	88.6	87.1	**84.2**	**82.7**
	Covington	MBL	83.9	79.7	86.6	84.8	79.9	78.1
		SVM	85.0	81.5	88.7	87.3	83.9	82.4

the languages. This baseline reflects the fact that dependency parsing is a fairly hard task for a computer. We have also checked if the well-formedness conditions are met. The parser output for all combinations is well-formed, i.e., every graph has a root, every graph is weakly connected, each node in the graph has at most one head, every graph is acyclic and projective. With this baseline and these well-formedness conditions we can conclude that we have succeeded to build a parser that produces well-formed dependency graphs with fairly high accuracy.

The experiment should also validate that it is possible to vary parsing algorithm, feature model and learning method without rebuilding the system. The experiment was performed by using an option file with all necessary options and these options were overridden by the command line flags and in that way the parser was controlled for all the runs without rebuilding it.

The results are interesting in themselves and the best accuracy for each feature model and for each language is in boldface. The lexicalized model Φ_5 outperforms the non-lexicalized model Φ_2 in all cases, and this is not surprising because the model Φ_5 uses more information to predict the next parser action. We will get back to the comparison of the two learning methods MBL and SVM in section 4.6, where we compare the methods in more detail, but the results presented in table 4.9 indicates that SVM performs better

than MBL at least for the lexicalized model Φ_5. The highest scores are achieved by using Nivre's arc-eager algorithm for Swedish and English, but for Chinese it is the arc-standard version that performs best.

If we look at the results language by language, the Swedish results have an obvious winner with the setup of SVM and Nivre's arc-eager algorithm with a highest score of 86.4% unlabeled and 82.6% labeled. Nivre's arc-standard algorithm performs several percentage points lower than the arc-eager for Swedish, but the results for Chinese are the opposite. For Chinese the best scores are performed using the arc-standard version with $AS_U = 84.2\%$ and $AS_L = 82.7\%$. One explanation could be that the Chinese language has properties that are hard to capture using Nivre's arc-eager algorithm. Maybe a more careful feature selection could eliminate this difference, but this has to be investigated in more detail and is beyond this thesis. The results for English indicate that Nivre's arc-eager algorithm achieves the highest scores at 88.9% unlabeled and 87.4% labeled. Another interesting observation is that the projective version of Covington's algorithm decreases the gap to the respective winner for each language by using the SVM learner.

4.5 Experiment II: LIBSVM Interface

Another interesting aspect to evaluate is the MaltParser specific settings for the SVM learner.[2] It is possible to reduce the learning and parsing time for the SVM learner by dividing the training instances into smaller sets based on some feature ϕ and/or on whether the top token has a head and train these sets separately (see section 2.3.3). A frequency threshold can be used for avoiding too small training sets, by pooling together all sets below a certain threshold t into a single set.

4.5.1 Experimental Setup

The investigation of the MaltParser specific SVM settings consists of three experiments. First, we will examine the four splitting strategies described in the Section 3.1.2: no splitting (ns), head splitting (hs), feature splitting (fs) and combined splitting (cs).

The second experiment will compare two features used for splitting the training instances. The two features are the part-of-speech of the token on top of the stack $(p(\sigma_0))$ and of the next input token $(p(\tau_0))$. We will finish up this section by experimenting with the frequency threshold t that determines

---

[2]We do not mean the LIBSVM parameters, which are interesting in themselves and will be explored more in detail in section 4.6

Table 4.10: Parsing accuracy for the four splitting strategies (S) used by the SVM implementation of MaltParser; two feature models (FM); two versions of Nivre's parsing algorithms (PA), where arc-eager is denoted NE and arc-standard NS. The accuracy is measured by the attachment score (AS) metrics; U: unlabeled, L: labeled.

FM	PA	S	Swedish		English		Chinese	
			AS_U	AS_L	AS_U	AS_L	AS_U	AS_L
Φ_2	NE	ns	82.1	75.3	81.4	78.4	73.1	70.8
		hs	82.0	75.1	81.3	78.2	73.0	70.7
		fs	**82.3**	**75.5**	**81.7**	**78.7**	73.2	70.9
		cs	81.9	75.1	81.6	**78.7**	73.0	70.8
	NS	ns	79.0	71.9	79.7	76.7	73.5	71.2
		fs	77.8	70.9	79.6	76.7	**73.6**	**71.4**
Φ_5	NE	ns	**86.8**	**83.1**	**89.2**	**87.7**	83.8	82.3
		hs	86.4	82.5	89.1	87.5	83.7	82.2
		fs	86.4	82.6	88.9	87.4	83.9	82.4
		cs	86.1	82.2	88.8	87.3	83.6	82.1
	NS	ns	85.6	81.6	88.6	87.1	**84.8**	**83.2**
		fs	84.6	80.5	88.6	87.1	84.5	83.0

if a certain feature value should be pooled together with others that are below the threshold t.

All three experiments are performed in combination with two feature models (Φ_2 and Φ_5) and two parsing algorithms (Nivre's arc-eager and Nivre's arc-standard).

4.5.2 Results and Discussion

The attachment scores for the first of the three experiments are presented in table 4.10, which shows the scores for two feature models, Φ_2 and Φ_5, and four splitting strategies for the arc-eager algorithm and two strategies for the arc-standard algorithm. For the latter it is not possible to divide the data based on whether the top of the stack has a head or not.[3] Although the gap between the strategies is not that big, the results indicate that the feature splitting strategy (fs) performs best for the unlexicalized model Φ_2, whereas no splitting ns is better for the more complex and lexicalized model Φ_5.

---

[3]The top token of the stack will never have a head when using Nivre's arc-standard algorithm (see section 2.3.1)

60

Table 4.11: Learning and parsing time for the four splitting strategies (S) used by the SVM implementation of MaltParser; two feature models (FM); two versions of Nivre's parsing algorithms (PA), where arc-eager is denoted NE and arc-standard NS; LT: learning time, PT: parsing time

FM	PA	S	Swedish		English		Chinese	
			LT	**PT**	**LT**	**PT**	**LT**	**PT**
Φ_2	NE	ns	12 min	3 min	15 h	1.5 h	3 h	47 min
		hs	10 min	2 min	10.5 h	1 h	2 h	26 min
		fs	48 s	12 s	**39 min**	5 min	25 min	7 min
		cs	**41 s**	**9 s**	46 min	**3 min**	**19 min**	**4 min**
	NS	ns	18 min	4 min	21 h	2 h	3,5 h	48 min
		fs	2 min	22 s	2 h	8 min	46 min	13 min
Φ_5	NE	ns	42 min	5 min	46 h	2 h	11 h	1 h
		hs	30 min	3 min	27 h	1 h	6.5 h	46 min
		fs	2 min	22 s	3 h	9 min	2 h	12 min
		cs	**1 min**	**14 s**	**2.5 h**	**4 min**	**1 h**	**8 min**
	NS	ns	1 h	5 min	62 h	2 h	12 h	1 h
		fs	3 min	28 s	5 h	9 min	1.5 h	11 min

A second observation is that the gap between fs and ns is more prominent for Swedish (0.4%) than for the other two languages for the Φ_5 model. Probably this can be explained by the fact that there are less data for Swedish. The arc-standard algorithm continues to give the best results for the Chinese data ($AS_U = 84.8\%$), but if we turn our attention to the arc-eager version and the Φ_5 model we can see an unexpected result in that fs is better than ns, although the difference is very small.

Table 4.11 shows the learning and parsing times for the same experiment. The most efficient strategy is when the parser combines the two splitting strategies hs and fs into cs, and this is not surprising because it gives the smallest datasets. There is one exception for the learning time for English but the gap is only 7 minutes. One of the big advantages of dividing the data according to the feature values is that it reduces both learning and parsing times by a factor of 15 for English and Swedish and approximately by a factor of 6 for Chinese.

This experiment shows that splitting the training instances into different sets and training on these separately reduces the learning and parsing time significantly without losing very much in parsing accuracy.

Table 4.12 presents the results for the second experiment, which investi-

Table 4.12: Parsing accuracy for the two features used for splitting the train-
ing instances; two feature models (FM); two versions of Nivre's parsing al-
gorithms (PA), where arc-eager is denoted NE and arc-standard NS. The
accuracy is measured by the attachment score (AS) metrics; U: unlabeled, L:
labeled.

FM	PA	PoS	Swedish		English		Chinese	
			AS_U	AS_L	AS_U	AS_L	AS_U	AS_L
Φ_2	NE	$p(\sigma_0)$	82.0	75.1	81.4	78.3	73.1	70.8
		$p(\tau_0)$	82.3	75.5	81.7	78.7	73.2	70.9
	NS	$p(\sigma_0)$	77.3	70.4	79.6	76.7	73.4	71.1
		$p(\tau_0)$	77.8	70.9	79.6	76.7	73.6	71.4
Φ_5	NE	$p(\sigma_0)$	86.0	82.1	88.5	87.0	83.7	82.2
		$p(\tau_0)$	86.4	82.6	88.9	87.4	83.9	82.4
	NS	$p(\sigma_0)$	84.6	80.6	88.5	87.0	84.6	83.1
		$p(\tau_0)$	84.6	80.5	88.6	87.1	84.5	83.0

gates the two splitting features $p(\sigma_0)$ and $p(\tau_0)$ for both versions of Nivre's
algorithm and the two feature models. The overall impression is that it is
better to divide the training instances based on the next input token $p(\tau_0)$,
rather than splitting on the token on top of the stack $p(\sigma_0)$, but there are
small indications that the opposite is true for Swedish and Chinese using the
arc-standard algorithm and the lexicalized model.

   In all previous experiments we have used the frequency threshold $t = 1000$,
which means that all sets consisting of less training instances than 1000 are
pooled together into a single model. The last experiment in this section will
vary this threshold for all the three languages and Nivre's arc-eager algorithm
and both feature models Φ_2 and Φ_5. The frequency threshold parameter t
was varied with the following values: 10, 50, 100, 500, 1000 and 1500.

   Because of the outcome of this experiment there is no table showing the
results. For English and Chinese there was no difference at all. There were
small differences for Swedish. The accuracy AS_U increases from 86.4% to
86.5% using $t = 1500$ for the lexicalized model, but for the unlexicalized
model $t = 1000$ is still the best result. One could argue that we should
not use this parameter at all, but if there exist feature values in the test
data that have not been seen during training then the parser will not know
how to predict the next parser action. This backoff model can handle these
situations, but another solution could be to simply provide the parser with a
default transition which is used when there is no model for the feature value.

Table 4.13: The parameter settings for MBL and SVM used in experiment III

	Swedish					English					Chinese				
	MBL		SVM			MBL		SVM			MBL		SVM		
	k	l	γ	C	r	k	l	γ	C	r	k	l	γ	C	r
Φ_1	6	1	.21	1	.6	10	2	.18	1	.3	4	2	.31	.5	.3
Φ_2	7	1	.21	1	.6	7	2	.23	1	0	4	2	.24	1	0
Φ_3	3	4	.21	1	0	10	3	.23	1	.3	6	5	.21	1	0
Φ_4	6	4	.21	.6	.3	6	3	.18	1	.3	10	4	.16	1	0
Φ_5	6	4	.40	.7	.3	7	3	.18	.5	.3	6	8	.20	.5	.5

4.6 Experiment III: Comparison of MBL and SVM

The previous experiments (see section 4.4.2) indicate that SVM is a better choice than MBL, especially for the feature model Φ_5. This section presents a more detailed comparison of SVM and MBL using Nivre's arc-eager algorithm. When comparing two machine learning methods it is important to investigate several feature models and make an extensive parameter optimization, as has been shown very clearly in recent research (Daelemans and Hoste 2002).

Comparative studies of learning algorithms are relatively rare in this context. Cheng et al. (2005b) report that SVM outperforms MaxEnt models in Chinese dependency parsing, using the algorithms of Yamada and Matsumoto (2003) and Nivre (2003), while Sagae and Lavie (2005) find that SVM gives better performance than MBL in a constituency-based shift-reduce parser for English.

4.6.1 Experimental Setup

The comparison involves three languages (Swedish, English and Chinese) and five feature models described in section 4.3. We will keep the parsing algorithm constant throughout this study, and Nivre's arc-eager algorithm will be used. As already noted, optimization of learning algorithm parameters is a prerequisite for meaningful comparison of different algorithms, although an exhaustive search of the parameter space is usually impossible in practice.

For MBL we have used the modified value difference metric (MVDM) and class voting weighted by inverse distance (ID) in all experiments, and performed a grid search for the optimal values of the number k of nearest

63

Table 4.14: Parsing accuracy on the final test set; five feature models (FM), two learning methods (LM). The accuracy is measured by the attachment score (AS) metrics; U: unlabeled, L: labeled.

FM	LM	Swedish		English		Chinese	
		$\mathbf{AS}_U$	$\mathbf{AS}_L$	$\mathbf{AS}_U$	$\mathbf{AS}_L$	$\mathbf{AS}_U$	$\mathbf{AS}_L$
Φ_1	MBL	75.3	68.7	*76.5	73.7	66.4	63.6
	SVM	75.4	68.9	76.4	73.6	66.4	63.6
Φ_2	MBL	81.9	74.4	81.2	78.2	73.0	70.7
	SVM	*83.1	*76.3	81.3	78.3	*73.2	*71.0
Φ_3	MBL	85.9	81.4	85.5	83.7	77.9	76.3
	SVM	86.2	*82.6	*86.4	*84.8	*79.7	*78.3
Φ_4	MBL	86.1	82.1	87.0	85.2	79.4	77.7
	SVM	86.0	82.2	*88.4	*86.8	*81.7	*80.1
Φ_5	MBL	86.6	82.3	88.0	86.2	81.1	79.2
	SVM	86.9	*83.2	*89.4	*87.9	*84.3	*82.7

neighbors and the frequency threshold l for switching from MVDM to the simple Overlap metric (cf. section 2.3.3).

The polynomial kernel of degree 2 has been used for all the SVM experiments, but the kernel parameters γ and r have been optimized together with the penalty parameter C and the termination criterion e. The training data are divided into smaller sets, according to the part-of-speech of the next token $p(\tau_0)$ in the current parser configuration and the frequency threshold t for separate classifiers is set to 1000 in all the experiments.

Table 4.13 shows the optimized value after an extensive search for each parameter and for each combination. The parameter optimization is done by using the development set for the English and the Chinese data and by doing the 9-fold cross validation for Swedish. The optimized values vary a lot and it is hard to draw general conclusions, more than that it is not trivial task to find the optimized values for each combination. Although these values are regarded here as optimized values, it is very likely that we could have found even more optimal values if we had continued the search.

4.6.2   Results and Discussion

Tables 4.14 and 4.15 show the parsing accuracy for the combination of three languages (Swedish, English and Chinese), two learning methods (MBL and SVM) and five feature models. For each combination we measure the *attachment score* (table 4.14) and the *exact match* (table 4.15). A significant

64

Table 4.15: Parsing accuracy on the final test set; five feature models (FM), two learning methods (LM). The accuracy is measured by the exact match score (EM) metrics; U: unlabeled, L: labeled.

FM	LM	Swedish		English		Chinese	
		EM_U	EM_L	EM_U	EM_L	EM_U	EM_L
Φ_1	MBL	16.0	11.4	9.8	7.7	14.3	12.1
	SVM	16.3	12.1	9.8	7.7	14.2	12.1
Φ_2	MBL	31.4	19.8	19.8	14.9	22.6	18.8
	SVM	*34.3	*24.0	19.4	14.9	22.1	18.6
Φ_3	MBL	37.9	28.9	26.5	23.7	26.3	23.4
	SVM	38.7	*32.5	*28.5	*25.9	*30.1	*25.9
Φ_4	MBL	37.6	30.1	29.8	26.0	28.0	24.7
	SVM	37.9	31.2	*33.2	*30.3	*31.0	*27.0
Φ_5	MBL	39.9	29.9	32.8	28.4	30.2	25.9
	SVM	40.7	*33.7	*36.4	*33.1	*34.5	*30.5

improvement for one learning method over the other is marked by an asterisk (*).

Independently of language and learning method, the most complex feature model Φ_5 gives the highest accuracy across all metrics. Not surprisingly, the lowest accuracy is obtained with the simplest feature model Φ_1. By and large, more complex feature models give higher accuracy, with one exception for Swedish and the feature models Φ_3 and Φ_4. It is significant in this context that the Swedish data set is the smallest of the three (about 20% of the Chinese data set and about 10% of the English one).

If we compare MBL and SVM, we see that SVM outperforms MBL for the three most complex models Φ_3, Φ_4 and Φ_5, both for English and Chinese. The results for Swedish are less clear, although the labeled accuracy for Φ_3 and Φ_5 are significantly better. For the Φ_1 model there is no significant improvement using SVM. In fact, the small differences found in the AS_U scores are to the advantage of MBL. By contrast, there is a large gap between MBL and SVM for the model Φ_5 and the languages Chinese and English. For Swedish, the differences are much smaller (except for the EM_L score), which may be due to the smaller size of the Swedish data set in combination with the technique of dividing the training data for SVM (cf. section 2.3.3).

Another important factor when comparing two learning methods is the efficiency in terms of time. Table 4.16 reports learning and parsing time for the three languages and the five feature models. The learning time corre-

65

Table 4.16: Learning and parsing time on the final test set; five feature models (FM), two learning methods (LM); LT: learning time, PT: parsing time

FM	LM	Swedish		English		Chinese	
		LT	PT	LT	PT	LT	PT
Φ_1	MBL	1 s	2 s	16 s	26 s	7 s	8 s
	SVM	40 s	14 s	1.5 h	14 min	1.5 h	17 min
Φ_2	MBL	3 s	5 s	35 s	32 s	13 s	14 s
	SVM	40 s	13 s	1 h	11 min	1.5 h	15 min
Φ_3	MBL	6 s	1 min	1.5 min	9.5 min	46 s	10 min
	SVM	1 min	15 s	1 h	9 min	2 h	16 min
Φ_4	MBL	8 s	2 min	1.5 min	9 min	45 s	12 min
	SVM	2 min	18 s	2 h	12 min	2.5 h	14 min
Φ_5	MBL	10 s	7 min	3 min	41 min	1.5 min	46 min
	SVM	2 min	25 s	1.5 h	10 min	6 h	24 min

lates very well with the complexity of the feature model and MBL, being a lazy learning method, is much faster than SVM. For the unlexicalized feature models Φ_1 and Φ_2, the parsing time is also considerably lower for MBL, especially for the large data sets (English and Chinese). But as model complexity grows, especially with the addition of lexical features, SVM gradually gains an advantage over MBL with respect to parsing time. This is especially striking for Swedish, where the training data set is considerably smaller than for the other languages.

66

Chapter 5

Conclusion

In this final chapter we summarize the main contributions of the thesis and point to promising directions for future work.

5.1 Main Results

We have designed an architecture for inductive labeled dependency parsing with a clean separation of parsing algorithms, feature models, and learning methods, so that these components can be varied independently of each other. Furthermore, the design makes it possible to reuse basic components so that they can be used in both learning and parsing. In this way we can reduce the amount of code and possible mistakes caused by having separate code for both phases. The *Parser* component is responsible for deriving the syntactic analysis for each sentence supported by sub-components that implement parsing algorithms. At every nondeterministic choice point, it asks the *Guide* component at parsing time to predict the next parser action by looking at the current parser state or at learning time to use the current parser state and the derived parser action to induce a parser model. The Guide is supported by discriminative learner interfaces that solve the classification problem at parsing time and induces classifiers at learning time.

This architecture has been realized in the MaltParser system, which can be seen as a parser generator tool for labeled dependency-based parsing. MaltParser is a terminal-based software package that is controlled either by an option file, by command line flags or by a combination of the two. The system can induce a parser model for any natural language for a given dependency-based treebank formatted in Malt-TAB. This parser model trained for a specific language can then be used to parse new unseen text data. Moreover, the system complies with the proposed architecture in that it is flexible and allows the user to specify arbitrary feature models using the feature specification language, and to choose parsing algorithms and learning methods by specifying options.

In the introductory chapter, we formulated three validation questions that

67

we should investigate. The first question deals with the implementation of the architecture and asks whether MaltParser meets the requirements of the architecture. After running 36 experiments, with combinations of two feature models, three algorithms, two classifiers and three languages, we can see that all results are above 70% both for unlabeled and labeled attachment score. The dependency graphs output by the parser satisfy the four basic constraints of well-formedness defined in section 2.2. These results support the conclusion that we have succeeded in implementing the architecture in accordance with the requirements.

The answer to the second question is that we can conclude that dividing the training data into smaller sets, according to the part-of-speech of the next token in the current parser configuration, substantially reduces both the learning and parsing time without significantly decreasing the parsing accuracy when using the SVM learner. There is a slight advantage in dividing the data based on the next token compared to the top token of the stack.

To answer the final question we have performed an empirical comparison of MBL (TiMBL) and SVM (LIBSVM) for deterministic dependency parsing, using treebank data from Swedish, English and Chinese and five feature models of varying complexity. The outcome shows that SVM gives higher parsing accuracy for complex and lexicalized feature models for English and Chinese, whereas for Swedish there is no significant improvement. The parsing efficiency is slightly better for SVM when using the most complex model, but for the other models it is the opposite. As expected, MBL has the fastest learning times for all models since it only stores the training data in a suitable representation, whereas SVM generalizes the data into a model.

For the licentiate thesis the goal was not to optimize the parser model for each language, which requires extensive work with feature selection in combination with parameter optimization. With that in mind, the aim was not to create a state of the art dependency-based parser. If we anyway compare the results to the state of the art in dependency parsing, we can report that for Swedish the attachment scores for model Φ_5 and Nivre's arc-eager algorithm are about .5 (unlabeled) and 1 (labeled) percentage points higher than the results reported for MBL by Nivre (2006) using an earlier version of MaltParser. The best results for English, where we also use the model Φ_5 and Nivre's arc-eager algorithm are about 3 percentage points below the results obtained with the parser of Charniak (2000) but less than 1 percentage point below the results obtained by Yamada and Matsumoto (2003) with a deterministic SVM-based parser. They are also about 1 percentage point better than the results reported for MBL by Nivre (2006). Finally, the highest accuracy for Chinese was obtained using the model Φ_5 and Nivre's arc-standard algorithm and is very close to the best reported results (about

68

85%) achieved with a deterministic classifier-based approach using SVM and preprocessing to detect root nodes (Cheng et al. 2005a), although these results are not based on exactly the same dependency conversion and data split as ours. It is also interesting that the arc-standard algorithm works better than arc-eager version, which is not the case for Swedish and English. This has to be investigated in more detail in future research.

5.2   Future Work

We can identify several issues that can be further explored for the doctoral thesis. The most important task for future work is to experiment with several more languages and to see if the architecture is suited for other languages. It is crucial that we investigate languages of different types. We have compiled a list of four major categories of future work:

1. **Efficiency**: One interesting factor for inductive labeled dependency parsing is the efficiency both during learning and parsing. This is very important, if a parser is to be useful in NLP applications. One way to increase the efficiency is to run more in parallel. For instance, we could allow sentences to be parsed simultaneously because they are not dependent on each other. Besides that, we could probably implement the SVM interface more efficiently by encoding the data in a smarter way without decreasing the accuracy together with new splitting strategies. Furthermore, it could be interesting to investigate combinations of two learning methods such that the accuracy is preserved and the efficiency is increased, for example, by combining a Naive-Bayes classifier with an SVM classifier.

2. **Optimization**: One experience we have made, when doing all the experiments, is that it is a non-trivial problem to find the best feature model in combination with the optimal parameters for the classifier. Doing a complete feature and parameter optimization is practically impossible. If we could find a good approach to finding a well adapted feature model together with near optimal parameters in reasonable time, it could be implemented in an automatic feature and parameter optimization tool.

3. **Parsing algorithms**: Another interesting factor could be to investigate the parsing algorithms in more detail. For example, why does Nivre's arc-standard algorithm perform better for Chinese, but not for English and Swedish? By extending the study to other languages,

69

maybe we could find general properties for a group of languages where a certain parsing algorithm is better suited.

4. **Nondeterminism**: Loosening the strictly deterministic approach to parsing may lead to higher accuracy, since we could explore several analyses at a more global level.

5. **Phrase structure**: Although dependency-based representations have gained more interest in recent years, the dominant kind of representation is still based on phrase structure. Therefore, it would be useful if we could find a way to convert the dependency-based output to phrase structure with high accuracy, preferably with a data-driven method that is not dependent on explicit rules.

Although this list of future work contains many interesting issues, it is probably too much to investigate all these in detail for the doctoral thesis. It is more likely that we will do a small survey and concentrate on the most promising ones.

70

References

Bies, A., M. Ferguson, K. Katz, and R. MacIntyre (1995). Bracketing guidelines for treebank II style, Penn Treebank project. University of Pennsylvania, Philadelphia.

Black, E., F. Jelinek, J. D. Lafferty, D. M. Magerman, R. L. Mercer, and S. Roukos (1992). Towards history-based grammars: Using richer models for probabilistic parsing. In *Proceedings of the 5th DARPA Speech and Natural Language Workshop*, pp. 31–37.

Boullier, P. (2003). Guided Earley parsing. In G. van Noord (Ed.), *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pp. 43–54.

Burges, C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery 2*(2), 121–167.

Cardie, C. (1993). Using decision trees to improve case-based learning. In *Proceedings of the 10th International Conference on Machine Learning*, pp. 25–32.

Chang, C.-C. and C.-J. Lin (2001). LIBSVM: A library for support vector machines.

Charniak, E. (2000). A maximum-entropy-inspired parser. In *Proceedings of the First Annual Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pp. 132–139.

Charniak, E. and M. Johnson (2005). Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 173–180.

Cheng, Y., M. Asahara, and Y. Matsumoto (2005a). Chinese deterministic dependency analyzer: Examining effects of global features and root node finder. In *Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing*, pp. 17–24.

Cheng, Y., M. Asahara, and Y. Matsumoto (2005b). Machine learning-based dependency analyzer for Chinese. In *Proceedings of the International Conference on Chinese Computing (ICCC)*.

Collins, M. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 16–23.

Collins, M. (1999). *Head-Driven Statistical Models for Natural Language Parsing*. Ph. D. thesis, University of Pennsylvania.

Collins, M. and N. Duffy (2005). Discriminative reranking for natural language parsing. *Computational Linguistics 31*, 25–70.

Collins, M., J. Hajič, L. Ramshaw, and C. Tillmann (1999). A statistical parser for Czech. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 505–512.

Corazza, A., A. Lavelli, G. Satta, and R. Zanoli (2004). Analyzing an Italian treebank with state-of-the-art statistical parsers. In *Proceedings of the Third Workshop on Treebanks and Linguistic Theories (TLT)*, pp. 39–50.

Cost, S. and S. Salzberg (1993). A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning 10*, 57–78.

Covington, M. A. (2001). A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pp. 95–102.

Daelemans, W. and V. Hoste (2002). Evaluation of machine learning methods for natural language processing tasks. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC)*, pp. 755–760.

Daelemans, W. and A. Van den Bosch (2005). *Memory-Based Language Processing*. Cambridge University Press.

Daelemans, W., A. Van den Bosch, and J. Zavrel (2002). Forgetting exceptions is harmful in language learning. *Machine Learning 34*, 11–43.

Daelemans, W., J. Zavrel, P. Berck, and S. Gillis (1996). A memory-based part of speech tagger generator. In E. Ejerher and I. Dagan (Eds.), *Proceedings of the Fourth Workshop on Very Large Corpora*, pp. 14–27.

Einarsson, J. (1976a). *Talbankens skriftspråkskonkordans*. Lund University, Department of Scandinavian Languages.

Einarsson, J. (1976b). *Talbankens talspråkskonkordans*. Lund University, Department of Scandinavian Languages.

Eisner, J. M. (1996). Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, pp. 340–345.

Eisner, J. M. (1999). Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 457–464.

Ejerhed, E. and G. Källgren (1997). Stockholm Umeå Corpus. Version 1.0. Produced by Department of Linguistics, Umeå University and Department of Linguistics, Stockholm University. ISBN 91-7191-348-3.

Hajič, J., B. Vidova Hladka, J. Panevová, E. Hajičová, P. Sgall, and P. Pajas (2001). Prague Dependency Treebank 1.0. LDC, 2001T10.

Hall, J. (2003). A probabilistic part-of-speech tagger with suffix probabilities. Master's thesis, Växjö University.

Hsu, C.-W., C.-C. Chang, and C.-J. Lin (2004). A practical guide to support vector classification. Technical report, Department of Computer Science and Information Engineering, National Taiwan University.

Hudson, R. A. (1984). *Word Grammar*. Blackwell.

Isozaki, H., H. Kazawa, and T. Hirao (2004). A deterministic word dependency analyzer enhanced with preference learning. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING)*, pp. 275–281.

Järborg, J. (1986). Manual för syntaggning. Technical report, Göteborg University, Department of Swedish.

Joachims, T. (1998). Text categorization with support vector machines. In *Proceedings of the 10th European Conference on Machine Learning (ECML'98)*, pp. 137–142.

Johnson, M., S. Geman, S. Canon, Z. Chi, and S. Riezler (1999). Estimators for stochastic "unification-based" grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 535–541.

Kay, M. (2000). Guides and oracles for linear-time parsing. In *Proceedings of the 6th International Workshop on Parsing Technologies (IWPT)*, pp. 6–9.

Kouchnir, B. (2004). A memory-based approach for semantic role labeling. In *Proceedings of the Eighth Conference on Computational Natural Language Learning*, pp. 118–121.

Kromann, M. T. (2003). The Danish Dependency Treebank and the DTAG treebank tool. In J. Nivre and E. Hinrichs (Eds.), *Proceedings of the Second Workshop on Treebanks and Linguistic Theories (TLT)*, pp. 217–220. Växjö University Press.

References

Kudo, T. and Y. Matsumoto (2000a). Japanese dependency structure analysis based on support vector machines. In *Proceedings of the Joint SIG-DAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC)*, pp. 18–25.

Kudo, T. and Y. Matsumoto (2000b). Use of support vector learning for chunk identification. In C. Cardie, W. Daelemans, and E. T. K. Sang (Eds.), *Proceedings of the 3rd Conference on Computational Natural Language Learning (CoNLL)*, pp. 142–144.

Kudo, T. and Y. Matsumoto (2001). Chunking with support vector machines. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Kudo, T. and Y. Matsumoto (2002). Japanese dependency analysis using cascaded chunking. In *Proceedings of the Sixth Workshop on Computational Language Learning (CoNLL)*, pp. 63–69.

Levy, R. and C. D. Manning (2003). Is it harder to parse Chinese, or the Chinese treebank? In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*.

Magerman, D. M. (1995). Statistical decision-tree models for parsing. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 276–283.

Marcus, M. P., B. Santorini, and M. A. Marcinkiewicz (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics 19*, 313–330.

Maruyama, H. (1990). Structural disambiguation with constraint propagation. In *Proceedings of the 28th Meeting of the Association for Computational Linguistics (ACL)*, Pittsburgh, PA, pp. 31–38.

McDonald, R., K. Crammer, and F. Pereira (2005). Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 91–98.

Nilsson, J., J. Hall, and J. Nivre (2005). MAMBA meets TIGER: Reconstructing a Swedish treebank from antiquity. In P. J. Henrichsen (Ed.), *Proceedings of the NODALIDA Special Session on Treebanks*.

Nivre, J. (2003). An efficient algorithm for projective dependency parsing. In G. van Noord (Ed.), *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pp. 149–160.

Nivre, J. (2006). *Inductive Dependency Parsing*. Springer.

74

Nivre, J. and J. Hall (2005). Maltparser: A language-independent system for data-driven dependency parsing. In M. Civit, S. Kübler, and M. Antònia marti (Eds.), *Proceedings of the Fourth Workshop on Treebanks and Linguistic Theories (TLT)*, pp. 137–148.

Nivre, J., J. Hall, and J. Nilsson (2004). Memory-based dependency parsing. In H. T. Ng and E. Riloff (Eds.), *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL)*, pp. 49–56.

Nivre, J., J. Hall, and J. Nilsson (2006). Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of the fifth international conference on Language Resources and Evaluation (LREC)*.

Nivre, J. and J. Nilsson (2005). Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 99–106.

Nivre, J., J. Nilsson, and J. Hall (2006). Talbanken05: A swedish treebank with phrase structure and dependency annotation. In *Proceedings of the fifth international conference on Language Resources and Evaluation (LREC)*.

Nivre, J. and M. Scholz (2004). Deterministic dependency parsing of English text. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING)*, pp. 64–70.

Oren, M., C. Papageorgiou, P. Sinha, E. Osuna, and T. Poggio (1997). Pedestrian detection using wavelet templates. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, pp. 193–199.

Osuna, E., R. Freund, and F. Girosi (1997). Training support vector machines: An application to face detection. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, pp. 130–136.

Ratnaparkhi, A. (1997). A linear observed time statistical parser based on maximum entropy models. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1–10.

Sagae, K. and A. Lavie (2005). A classifier-based parser with linear runtime complexity. In *Proceedings of the 9th International Workshop on Parsing Technologies (IWPT)*, pp. 125–132.

Teleman, U. (1974). *Manual för grammatisk beskrivning av talad och skriven svenska*. Studentlitteratur.

75

Van den Bosch, A., S. Canisius, W. Daelemans, I. Hendrickx, and E. Tjong Kim Sang (2004). Memory-based semantic role labeling: Optimizing features, algorithm, and output. In *Proceedings of the Eighth Conference on Computational Natural Language Learning*, pp. 102–105.

Vapnik, V. (1979). Estimation of dependences based on empirical data. Technical report, Nauka, Moscow.

Vapnik, V. (1998). *Statistical Learning Theory*. John Wiley and Sons, New York.

Veenstra, J. and W. Daelemans (2000). A memory-based alternative for connectionist shift-reduce parsing. Technical Report ILK-0012, University of Tilburg.

Vural, V. and J. G. Dy (2004). A hierarchical method for multi-class support vector machines. *ACM International Conference Proceeding Series 69*, 105–113.

Xue, N., F.-D. Chiou, and M. Palmer (2002). Building a large-scale annotated chinese corpus. In *The 19th International Conference on Computational Linguistics (COLING)*.

Xue, N., F. Xia, F.-D. Chiou, and M. Palmer (2004). The Penn Chinese Treebank: Phrase structure annotation of a large corpus. *Natural Language Engineering 11*(2), 207–238.

Yamada, H. and Y. Matsumoto (2003). Statistical dependency analysis with support vector machines. In G. van Noord (Ed.), *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pp. 195–206.

76

Växjö University

School of Mathematics and Systems Engineering