



UPPSALA
UNIVERSITET

MaltParser and LIBLINEAR

*Transition-based dependency parsing with linear
classification for feature model optimization*

Sofia Cassel

Uppsala University
Department of Linguistics and Philology
Språkteknologiprogrammet
(Language Technology Programme)
Master's thesis in Computational Linguistics

December 7, 2009

Supervisor:
Marco Kuhlmann

Abstract

In this thesis, MaltParser has been extended with an interface to a software for large-scale linear classification (LIBLINEAR). This combination was then used for learning and parsing with four different treebanks (Slovene, Danish, Arabic, and Turkish).

The initial tests compared the accuracy of MaltParser using LIBLINEAR to that when using LibSVM (MaltParser's default classifier) with its linear algorithm. The results are significantly different in favor of LibSVM for two of the four treebanks (Danish and Arabic).

The LIBLINEAR classifier was then used for optimizing a feature model for each of the four treebanks. These results showed a significant improvement compared to the baseline feature model both when using only LIBLINEAR, and when using the optimized feature model with LibSVM. Learning and parsing times are much smaller when using LIBLINEAR, though accuracy is not quite as high as that of LibSVM's polynomial kernel.

In conclusion, using LIBLINEAR for feature model optimization is a good idea, since it takes advantage of the speed of LIBLINEAR while not sacrificing accuracy when using LibSVM's polynomial kernel for the final feature model. This also indicates that the MaltParser/LIBLINEAR combination should be explored further, and preferably with larger treebanks.

Sammandrag

I denna uppsats har MaltParser utökats med ett interface till LIBLINEAR, ett verktyg för storskalig linjär klassificering av data. Denna kombination användes sedan för inläring och parsning tillsammans med fyra olika trädbanker (slovenska, danska, arabiska och turkiska).

De inledande testerna jämförde korrektheten för två olika klassificeringsmetoder i MaltParser, dels standardklassificeraren LibSVM med en linjär algoritm, dels den nya klassificeraren LIBLINEAR. Resultaten visade en signifikant skillnad till fördel för LibSVM för två av de fyra trädbankerna (danska och arabiska).

Sedan användes LIBLINEAR för att optimera en feature-modell för var och en av de fyra trädbankerna. Resultaten visade en signifikant förbättring jämfört med en baseline-modell, både vid användning av bara LIBLINEAR och när man använder den färdiga feature-modellen tillsammans med LibSVM. Det går också åt mycket mindre tid för inläring och parsning när man använder LIBLINEAR. Dock blir systemets korrekthet bättre när man använder LibSVM med polynomisk kärna.

Slutsatsen är att det är en god idé att använda LIBLINEAR för att optimera feature-modeller med MaltParser. Detta drar nämligen nytta av de korta körningstiderna som fås med LIBLINEAR samtidigt som man kan använda LibSVM, och därmed få bättre korrekthet, för den slutliga feature-modellen. Resultaten indikerar också att kombinationen MaltParser/LIBLINEAR borde utforskas ytterligare och även med större trädbanker.

Contents

1	Introduction	5
1.1	Purpose	6
2	Background	7
2.1	Dependency graphs	7
2.2	Dependency parsing	8
2.3	Inductive dependency parsing	9
2.3.1	Parsing algorithms	9
2.3.2	Transition system	9
2.3.3	Oracle function	12
2.3.4	Feature models	12
2.3.5	Discriminative machine learning	13
3	Method	15
3.1	Machine learning packages	15
3.2	Trebanks	16
3.3	LIBLINEAR plugin	16
3.4	Initial tests	16
3.5	Feature model optimization	18
3.5.1	Backward selection	18
3.5.2	Forward selection	19
3.5.3	Optimization	19
4	Results and discussion	20
4.1	Initial tests	20
4.2	Feature model optimization	21
4.2.1	Accuracy	22
4.2.2	Learning and parsing times	22
5	Conclusions	24
	Bibliography	25

Acknowledgements

I would like to express my gratitude to Marco Kuhlmann for supervising this thesis, for his advice and constant encouragement.

Thanks also to Johan Hall for helping with the Java code.

Finally, I would like to thank Per-Erik Malmström for answering all my LaTeX questions, Julia Hammar for many interesting discussions, and Carl Hallberg, for being my tech support (including drawing Figure 2.2).

1 Introduction

Dependency parsing is the process of mapping a given input sentence to a syntactic representation, in the form of a dependency graph. It has become a popular technique for parsing natural language in the form of sentences. Within dependency parsing, transition-based systems have been shown to obtain good results for several different languages, for example in the CoNLL¹ shared tasks of 2006 (Buchholz and Marsi, 2006) and 2007 (Nivre et al., 2007).

The basic idea of *transition-based dependency parsing* is that a parser makes a series of transitions in order to get from an initial state, or configuration, to a terminal one. In the initial state, none of the words in the input text have been processed yet. In the terminal state, all the words have been processed and an (ideally, correct) analysis of the input text has been produced by the parser.

In order to be able to produce correct analyses of sentences in a given language, the system needs to be trained first. Some method for inductive learning is used, meaning basically that the system learns from a set of examples, in this case correctly analyzed sentences. The system applies this knowledge to new, previously unseen sentences and is then hopefully able to analyze these correctly.

When learning by example, a feature model determines what aspects, or features, of the learning examples that the system will consider. Such features may, for example, be the part-of-speech or the lemma of a word in the input sentence.

One available system for transition-based dependency parsing is MaltParser. It relies completely on inductive learning, using a classifier trained on an annotated treebank to derive a dependency graph. MaltParser is one of the systems that performed very well in the CoNLL shared tasks of 2006 (Buchholz and Marsi, 2006) and 2007 (Nivre et al., 2007).

MaltParser v 1.2 comes packaged with an interface to the machine learning package LibSVM. When using this classifier, MaltParser can achieve high accuracy scores; on the other hand, learning and parsing is quite time- and memory-consuming, especially when dealing with larger treebanks. For example, running MaltParser in training mode on the Turkish treebank (Atalay et al. (2003); Oflazer et al. (2003)) takes a little more than an hour (58,000 tokens). Training a model on the very large Czech treebank (1,249,000 tokens, Böhmová et al. (2003)), which is more than twenty times the size of the Turkish treebank, takes over a week using a modern computer.

The problem of learning and parsing times becomes even more prominent when dealing with feature model optimization. Feature model optimization is the process of determining what features should be included in a feature

¹Conference on Computational Natural Language Learning

model in order to obtain an accuracy as high as possible. This requires many consecutive executions of the MaltParser system in learning and parsing mode, which makes it a very time-consuming task, especially with larger treebanks.

One solution to the problem is to split the training data into smaller pieces, training the system on one piece at a time. Another, more interesting solution, which is explored in this thesis, is the idea of using a different classifier better suited for use with large treebanks.

1.1 Purpose

In this thesis, MaltParser v 1.2 has been extended with an interface to LIBLINEAR, a machine learning package developed especially for classifying large amounts of data.

First, the MaltParser/LIBLINEAR combination had to be tested. Which of the classifiers in the LIBLINEAR package would be the best choice, and how would the new classifier perform compared to the default one? It would, for example, be reasonable to assume a lower accuracy, since LIBLINEAR deals only with linear classification, as opposed to the default classifier LibSVM, which is capable of performing both linear and nonlinear classification.

The second experiment concerned feature model optimization. Can this process be improved in any way through the use of a different classifier? What happens when we try to use the LIBLINEAR classifier in this process, and can its use be justified with respect to learning and parsing times as well as accuracy?

2 Background

In natural language parsing, the task of the parser is to produce a correct syntactic analysis of some sample of text. One way of syntactically representing a sentence is through a *dependency graph*.

2.1 Dependency graphs

A dependency graph expresses links between individual words, rather than breaking the sentence up into smaller phrases as in a constituency tree. Figure 2.1 shows an example of a dependency graph.

The links, or arcs, denote dependency relations between two words, called *head* and *dependent*. The arc labels signify the type of dependency relation; in Figure 2.1 below, for example, there is an arc labeled NMOD from the second to the first node. This arc itself specifies that the first node is the dependent of the second node, while the label indicates that the relation is that of a nominal modifier.

For the sentence $x = (w_0, w_1, \dots, w_n)$, the dependency graph is thus defined as $G = (V, A)$ (Nivre, 2008) where:

- $V = \{0, 1, \dots, n\}$ is the set of nodes, each corresponding to the linear position of a word in the sentence x . The node 0 is artificial and corresponds to an artificial ROOT word inserted at the beginning of the sentence in position 0. For the purpose of this thesis, and following Nivre (2006), all nodes corresponding to actual words in the sentence x are called *token nodes*.
- $A \subseteq V \times L \times V$ is the set of labeled, directed arcs between the nodes. Each arc is a triple (i, l, j) where i and j are nodes, and $l \in L$ the arc label.

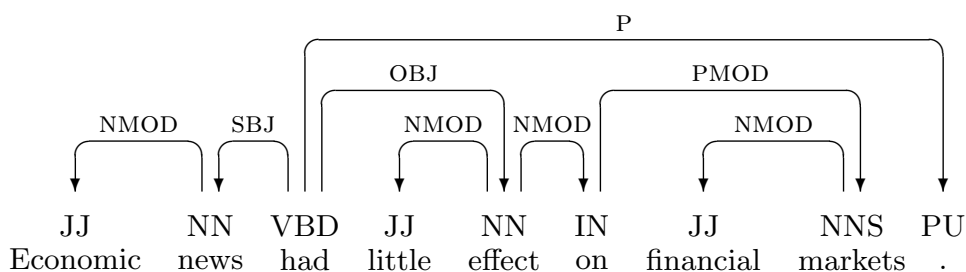


Figure 2.1: Dependency graph for an English sentence

In the dependency graph shown in Figure 2.1, not only arcs have labels, but the words (including punctuation) making up the sentence are also marked. Here, the example sentence is taken from a treebank where each word is tagged; JJ, NN, VBD etc. are all examples of part-of-speech tags (POS tags). The tags, if any, are thus dependent on the treebank used. For the purpose of this thesis, a *token* is defined as a word and any associated tags.

A dependency graph is defined as *well-formed* if it satisfies some basic requirements (Nivre, 2008):

- Each node is allowed at most one head (another node). This is also called the *single-head constraint*.
- The node 0 must be the root, which means that there can be no arc $(i, l, 0)$ where i is some token node and l is the arc label.

Projectivity is another constraint that may or may not be imposed upon the dependency graph. In order for the graph to be projective, for every arc $(i, l, j) \in A$ and every node k , if $i < k < j$ or $i > k > j$, there must also be some subset of arcs forming a path from i to k .

2.2 Dependency parsing

The task of a system for dependency parsing may be taken as that of mapping a given input sentence to a well-formed dependency graph. Within text parsing in general, and here dependency parsing in particular, two different approaches can be distinguished (Nivre, 2006): grammar-driven and data-driven parsing.

Grammar-driven parsing requires a formal grammar defining the well-formed analyses of the parsed language. An appropriate parsing algorithm is applied to the input text, producing an analysis which is allowed by the grammar and therefore considered correct.

Data-driven parsing is the focus of this chapter. It is a very versatile approach to syntactic parsing since it can be applied to any language as long as there is an annotated treebank available. Data-driven parsing requires no pre-existing grammar; in order to derive correct analyses of sentences, it applies to the treebank some method for *inductive learning*. Inductive learning is a form of 'learning by example', and as such, is dependent on the number of examples available for training.

McDonald and Nivre (2007) describe the two most widely used models for data-driven dependency parsing: *graph-based* and *transition-based* parsing.

- In *graph-based* dependency parsing, for each input sentence all possible, valid dependency graphs are assigned a score. Some method for inductive learning is used to determine how to score each dependency graph. The system then performs a search for the highest-scoring dependency graph. Depending on the parsing algorithm, projective and/or non-projective structures can be captured.
- In *transition-based* dependency parsing, exemplified by MaltParser, the parser instead makes a series of transitions to get from an initial configuration to a terminal configuration, representing a finished dependency

graph. Inductive learning is used to guide the parser in deciding what transition to choose in cases where more than one transition is possible.

While systems based on the two respective models behave differently and tend to make different types of errors in deriving dependency graphs, their performance is quite similar (McDonald and Nivre, 2007). Combinations of the models have also been explored, with one parser generating features for the other (Nivre and McDonald, 2008).

2.3 Inductive dependency parsing

MaltParser has been developed within the framework of *inductive dependency parsing* (Nivre, 2006). This framework can be viewed as based on three elements, all of which will be discussed further below:

- (1) deterministic parsing algorithms,
- (2) history-based feature models, and
- (3) discriminative machine learning (here exemplified by *support vector machines*).

2.3.1 Parsing algorithms

MaltParser includes two groups of parsing algorithms, Nivre’s (2003) and Covington’s (2000), respectively. Each group contains two parsing algorithms. In the Nivre group, which is the focus here, the algorithms are called *arc-eager* and *arc-standard*, respectively. They both process input from left to right, making use of a stack for partially processed nodes, and a queue for tokens remaining in the input. More so than the arc-eager algorithm, used in the experiments of this thesis, the arc-standard version is similar to a basic shift-reduce algorithm.

2.3.2 Transition system

For transition-based parsing, a transition system consists of a set of parser configurations and the transitions between them.

The transition system defines:

- a set of *parser configurations* C ,
- an *initialization function* c_s ,
- a set of *terminal configurations* $C_t \subseteq C$, and
- a set of *transitions* T .

Parser configuration

For Nivre’s algorithms, each of the parser configurations $c \in C$ for a given input sentence $x = (w_0, w_1, \dots, w_n)$ can be characterized as $c = (\sigma, \tau, A)$ (Nivre, 2008). The parser configuration is explained in the following way:

- σ is a stack of partially processed token nodes.
- τ is the list of nodes representing the remaining words in the input sentence.
- A is a set of dependency arcs, such that the dependency graph for the sentence x is defined as $G = (V, A)$ where $V = [0, 1, \dots, n]$.

Each non-terminal parser configuration for the sentence $x = (w_0, w_1, \dots, w_n)$ thus defines a (partially-built) dependency graph.

Initialization function

The initialization function c_s maps the sentence $x = (w_0, w_1, \dots, w_n)$ to a configuration with $\tau = [1, \dots, n]$ and where the stack σ is empty. Initially, all token nodes are also set as dependents of the root node, since they have not yet been attached to a head.

Terminal configurations

A parser configuration is called terminal if τ is empty, i.e. if the parser has reached the end of the sentence and there are no more words to be parsed. A sentence may have more than one terminal configuration.

Transitions

Nivre's arc-eager algorithm provides the following four transitions, each available for every dependency type l (Nivre, 2008):

- **Right-arc(l)**
Creates an arc (i, l, j) with the dependency type l from the node i on the top of the stack to the next node j in the input list, pushing j onto the stack. In order to satisfy the *single-head* constraint, the node j may not already have a head. The arc (i, l, j) is added to the set A .
- **Left-arc(l)**
Creates an arc (j, l, i) with the dependency type l from the next input node j to the token node i on the top of the stack, popping i from the stack. In order to satisfy the *single-head* constraint, the node j may not already have a head. Since the root node 0 may not have a head, i cannot be the head. The arc (j, l, i) is added to the set A .
- **Reduce**
Pops the stack, removing the topmost node i . This is permitted only when the head of the topmost token node is another token node and not the root node. It is not allowed to pop the topmost node unless it has been assigned a head, because it cannot be attached later.
- **Shift**
Shifts (pushes) the next input token node onto the stack. This is permitted as long as there are token nodes remaining to be processed.

Using this algorithm, it is possible to step through the analysis of the sentence in the Figure 2.1 dependency graph. Starting at the beginning of the sentence, the transition sequence would be:

- (1) **Shift**, the only possible transition out of the initial state. 'Economic' is now on top of the stack.
- (2) **Left-arc(NMOD)**, creating an arc from 'news' to 'Economic' and popping 'Economic' off the stack. The stack is now empty.
- (3) **Shift**, pushing 'news' onto the top of the stack.
- (4) **Left-arc(SBJ)**, creating an arc from 'had' to 'news' and popping 'news' off the stack. The stack is now empty.
- (5) **Shift**, pushing 'had' onto the top of the stack.
- (6) **Shift**, pushing 'little' onto the top of the stack.
- (7) **Left-arc(NMOD)**, creating an arc from 'effect' to 'little' and popping 'little' off the stack. Again 'had' is on top of the stack.
- (8) **Right-arc(OBJ)**, creating an arc from 'had' to 'effect' and pushing 'effect' onto the top of the stack.
- (9) **Right-arc(NMOD)**, creating an arc from 'effect' to 'on' and pushing 'on' onto the top of the stack.
- (10) **Shift**, pushing 'financial' onto the top of the stack.
- (11) **Left-arc(NMOD)**, creating an arc from 'markets' to 'financial' and popping 'financial' off the stack. 'on' is now on top of the stack.
- (12) **Right-arc(PMOD)**, creating an arc from 'on' to 'markets' and pushing 'markets' onto the top of the stack.
- (13) **Reduce**, popping 'markets' off the stack. 'on' is now on top of the stack.
- (14) **Reduce**, popping 'on' off the stack. 'effect' is now on top of the stack.
- (15) **Reduce**, popping 'effects' off the stack. 'had' is now on top of the stack.
- (16) **Right-arc(P)**, creating an arc from 'had' to the next (and final) token node in the input, '.', pushing it onto the top of the stack. The input list τ is now empty and the parser has reached a terminal configuration.

Nivre's arc-standard algorithm shares the Shift and Left-arc transitions with the arc-eager version, but the Reduce transition is not available, and the Right-arc transition is slightly different: it moves the node on top of the stack i to the list of remaining input nodes τ , replacing j as the next node.

2.3.3 Oracle function

Two requirements of a dependency parser are *robustness* and *disambiguation*. Robustness means that the parser needs to be able to assign each input sentence at least one analysis. Disambiguation means that the parser needs to be able to assign each input sentence at most one analysis. Combined, this leads to the parser being allowed to assign exactly one analysis to each input sentence.

The transition system described above is in itself nondeterministic. At a given parser configuration, more than one transition may be applicable. In order to fulfill the robustness and disambiguation requirements, the parser therefore needs some additional mechanism for deciding which transition to apply. Such a mechanism can be called an *oracle*.

In theory, the oracle (Kay, 2000) is a function mapping nonterminal configurations to transitions, or $o : C_n \rightarrow (C_n \rightarrow C)$ where $C_n \rightarrow C$ is the transition function t , so the oracle can be rewritten as $o : C_n \rightarrow t$. When the oracle function is applied to any nonterminal configuration, it returns the one and only correct transition out of that configuration.

In practice, the oracle function can only be approximated, for example by inducing a classifier on a treebank of annotated sentences in the language to be parsed.

2.3.4 Feature models

When training a classifier in order to approximate an oracle, the goal is to have the classifier derive the correct transitions from the annotated sentences in the treebank. In order to do this, a one-to-one mapping is defined from an input string and its dependency graph to a sequence of parser transitions. Every transition is dependent on all previous transitions made by the parser, and the *history* contains complete information about these.

The idea of a *history-based feature model* is to make the learning problem tractable by extracting only certain relevant parts of the history. A theoretically ideal history-based feature model may therefore be defined as one that includes all parts of the history that contribute to the parser accuracy, while at the same time excluding all parts of the history that do not.

The feature model can be defined as a set of feature functions, where each function corresponds to an attribute of the (current) parser state. When applying these feature functions to the parser state at any given time, a sequence of features is returned. This sequence of features is called a feature vector, and it is then used by the classifier for predicting the next transition.

It is common for feature models to include both static and dynamic features. Dynamic features may be different depending on the parser state when the feature function was applied; static features remain the same throughout parsing. Examples of static features are the part-of-speech tag or word form of a token. Dynamic features, for example dependency types (arc labels) are determined at some point during the parsing process and thus are not the same for every parser state.

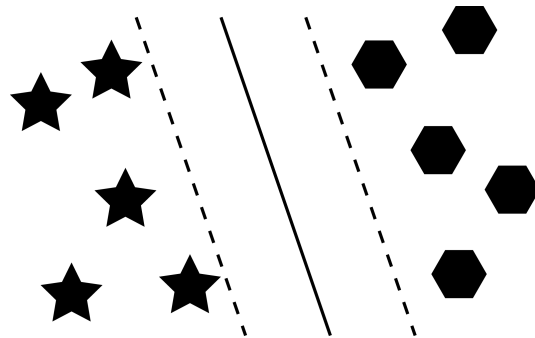


Figure 2.2: Maximum-margin strategy.

2.3.5 Discriminative machine learning

The task of classification generally means predicting a class y given an observation x , or in other words, deciding what class y that the observation x belongs to. In a very basic version of binary classification, for each x the classifier would simply return the value 0 or the value 1, representing the class y .

In transition-based dependency parsing, classification is used to determine the parser decision whenever several transitions are possible out of a given parser configuration. The class y thus represents a parser decision, whereas the observation x is the parser history. For every observation x , the classifier assigns a score to each class y , and predicts the highest-scoring class as the class that x belongs to.

Support vector machines

Support vector machines are commonly used for binary classification. The basic idea of SVM, introduced by Vapnik (1995) is to separate the two classes by as wide a margin as possible, assuming that the data is linearly separable. Linear separation can be exemplified by drawing a line on a graph of the training data, separating the two classes, given that the input feature vector is of dimensionality 2. An example of this is seen in Figure 2.2. If the input is of a higher dimensionality, the two classes may be separated by a hyperplane.

SVM can be used to classify data that is not linearly separable. A cost, or penalty parameter may be introduced in order to allow for some misclassification. This C parameter controls the tradeoff between allowing for some data points to be misclassified and enforcing the margin between the classes. A higher C value enforces a harder margin between the classes. This can result in so-called *overfitting*, obtaining a more accurate model for that particular set of training data, but one that may not perform well when used with another data set. A lower C value means more misclassifications are allowed, perhaps resulting in a lower accuracy. It is thus important to select an appropriate value for this parameter in order to achieve good accuracy without overfitting the model.

Further, the so-called kernel trick may be used to transform the linear SVM algorithm into one that is able to handle non-linearly separable data. The kernel trick means replacing every calculation of a dot product between two vec-

tors with a kernel function. There are several such functions; some examples are the polynomial, the radial basis, and the sigmoid kernels. The result is the input feature vectors mapped to a higher-dimensional space, where the data is linearly separable.

Multi-class classification is also possible with SVM. Two methods for this, using many binary classifiers, are *one-versus-one* and *one-versus-all*. With the one-versus-all method, for n classes as many classifiers are trained in order to separate each class from the rest. With the one-versus-one method, one classifier is trained for each pair of classes. A voting system may be used to discriminate between the classes.

3 Method

In this chapter, first the materials used in experiments will be described: two machine learning packages, four treebanks (Slovene, Danish, Arabic, and Turkish), and the MaltParser plugin. This is followed by an account of the experiments performed.

3.1 Machine learning packages

Two machine learning packages have been used in all the experiments in this thesis: LibSVM and LIBLINEAR.

LibSVM is a package for support vector classification, using the support vector machines technique. There are a number of options available to the user, a few of which will be discussed here. LibSVM includes several kernels, including RBF (radial basis function), a polynomial and a linear kernel. Here, only the polynomial kernel of degree 2 (quadratic) and the linear kernel have been used. When using LibSVM, the SVM-type option has always been set to 0, corresponding to C-SVC (Hsu et al., 2008). The version used is 2.88, which is also included with the MaltParser v. 1.2 distribution.

LIBLINEAR is a package for linear classification only, with no kernel options. It supports linear SVM and logistic regression, and for multi-class classification also a method developed by Crammer and Singer (2000). These are implemented as different *solver type* options for the user (Fan et al., 2008). The LIBLINEAR version used in these experiments is 1.33.

It should be noted that the algorithms behind LibSVM and LIBLINEAR are entirely different. While LIBLINEAR performs linear classification directly, the LibSVM algorithm always employs a kernel function. Thus in order to implement linear classification, the LibSVM classifier takes advantage of a special case of the RBF kernel, calling it a linear kernel (Hsu et al., 2008).

Common to LIBLINEAR and LibSVM are the C and ϵ parameters, the only other options dealt with in this thesis. The C parameter is equivalent to the penalty parameter discussed in ??, whereas the ϵ parameter defines the termination criterion for the classifier's optimization problem.

Both packages have also been developed by the same team, Hsu Chang and Lin, and are freely available for download from the LibSVM¹ and LIBLINEAR webpages², respectively. LibSVM contains source code in both C++ and Java, but the source code for LIBLINEAR is in C/C++ only. There is, however, a Java version of the LIBLINEAR package available; developed by Waldvogel, it

¹<http://www.csie.ntu.edu.tw/~cjlin/libsvm>

²<http://www.csie.ntu.edu.tw/~cjlin/liblinear>

is also downloadable from the LIBLINEAR webpage, and it is this version of the package that has been used in the subsequent experiments.

3.2 Treebanks

The treebanks used for the experiments below contain annotated sentences in Slovene, Danish, Arabic, and Turkish, respectively. As seen in table 3.1, the treebanks are of varying sizes. The Slovene Dependency Treebank (Džeroski et al., 2006) is the smallest, followed by the Prague Arabic Dependency Treebank (Smrž et al. (2004); Smrž et al. (2002)) and the Turkish Metu-Sabancı treebank (Atalay et al. (2003); Oflazer et al. (2003)). The Danish Dependency Treebank (Kromann, 2003) is the largest data set. All the treebanks are in CoNLL format³, and were those used in the CoNLL-X shared task on multilingual dependency parsing (Buchholz and Marsi, 2006).

	Slovene	Danish	Arabic	Turkish
Number of tokens	29,000	94,000	54,000	58,000
Number of sentences	1,500	5,200	1,500	5,000

Table 3.1: Treebanks

3.3 LIBLINEAR plugin

Making it possible for MaltParser to use the LIBLINEAR package for learning and parsing was the first goal of this project. The most obvious way to approach this was to create an interface to the new machine learning software, taking advantage of the MaltParser plugin management system which allows developers to add functionality through self-controlled plugins which are loaded automatically on startup of the system (Hall, 2006). Since the LIBLINEAR package is so similar to the LibSVM package, the current LibSVM interface was modified so that it would work with LIBLINEAR instead, and packaged as a MaltParser plugin.

When modifying the LibSVM interface, certain differences in the structure of the machine learning packages were discovered. One issue that needed to be addressed was the k-best list option. While LibSVM is able to produce a list of predicted parser transitions, LIBLINEAR is not; for that reason the k-best list was disabled in the plugin.

3.4 Initial tests

The initial tests with MaltParser and the newly added plugin were performed in order to assure that the plugin was working properly and that the parser would be able to obtain a reasonable accuracy with LIBLINEAR.

³<http://nextens.uvt.nl/conll/#dataformat>

For each of the four treebanks Slovene, Danish, Arabic, and Turkish, Malt-Parser was run in learning and then parsing mode using the LIBLINEAR plugin. Initially, all the parameters of LIBLINEAR were set to their default value.

The MaltParser options were left unchanged from their default values, except for the feature models. Specific to each language and treebank, the feature models were those used with MaltParser in the CoNLL-X Shared Task on multilingual dependency parsing. The feature models for all four treebanks are shown in table 3.2.

Feature	Slovene	Danish	Arabic	Turkish
POSTAG, Stack[0]	+	+	+	+
POSTAG, Input[0]	+	+	+	+
POSTAG, Input[1]	+	+	+	+
POSTAG, Input[2]	+	+	+	+
POSTAG, Input[3]		+	+	
POSTAG, Stack[1]	+	+	+	+
POSTAG, succ(Stack[0])	+			
POSTAG, pred(Input[0])	+	+	+	
DEPREL, rsib(ldep(Stack[0]))			+	
DEPREL, Stack[0]	+	+	+	+
DEPREL, ldep(Stack[0])	+	+		+
DEPREL, rdep(Stack[0])	+	+		+
DEPREL, ldep(Input[0])	+	+		+
DEPREL, rsib(ldep(Stack[0]))	+			
FORM, Stack[0]	+	+	+	+
FORM, Input[0]	+	+	+	+
FORM, Input[1]	+	+	+	+
FORM, head(Stack[0])	+	+	+	
FORM, pred(Input[0])		+	+	
FORM, rdep(Stack[0])		+		
FORM, ldep(Input[0])		+		
FORM, lsib(rdep(Stack[0]))			+	
Split(FEATS, Stack[0]),\	+	+	+	+
Split(FEATS, Input[0]),\	+	+	+	+
Split(FEATS, Input[2]),\	+			
Split(FEATS, pred(Input[0]),\		+		
CPOSTAG, Stack[0]	+	+	+	+
CPOSTAG, Input[0]	+	+	+	+
CPOSTAG, Stack[2]	+			
CPOSTAG, pred(Input[0])		+		
CPOSTAG, ldep(head(Stack[0]))			+	
LEMMA, Stack[0]	+		+	+
LEMMA, Input[0]			+	+

Table 3.2: Initial feature models

The goal of these initial tests was to find the solver type that would generate the best accuracy. In order to do this, a series of tests was carried out with the five different solver types being matched with different values of the penalty

parameter C.

In learning mode, MaltParser was run using the gold standard treebanks as input. In parsing mode, the program was run using the test set provided with the gold standard treebank as input. MaltEval (Nilsson and Nivre, 2008) was used for evaluation. The evaluation metrics were LAS (labeled attachment score) per token, or the proportion of tokens assigned the correct head as well as the correct label when compared to the gold standard data.

MaltParser was also run using the LibSVM learner and the linear kernel in order to find for each treebank the best value of the C parameter. Accuracy was again measured as LAS using MaltEval, and the best LAS compared to the results obtained with the LIBLINEAR plugin.

3.5 Feature model optimization

Using as a baseline the example feature model supplied with the MaltParser package, feature models for the selected treebanks (Slovene, Danish, Turkish, and Arabic) were optimized using only the LIBLINEAR learner. The baseline feature model, containing 14 features, is shown in table 3.3.

Feature
POSTAG, Stack[0]
POSTAG, Input[0]
POSTAG, Input[1]
POSTAG, Input[2]
POSTAG, Input[3]
POSTAG, Stack[1]
DEPREL, Stack[0]
DEPREL, ldep(Stack[0])
DEPREL, rdep(Stack[0])
DEPREL, ldep(Input[0])
FORM, Stack[0]
FORM, Input[0]
FORM, Input[1]
FORM, head(Stack[0])

Table 3.3: Baseline feature model

In performing the feature model optimization, *backward selection* as well as *forward selection* was used.

3.5.1 Backward selection

Backward selection means starting by fitting a model with all variables of interest. The variable which contributes the least to the model, or whose inclusion does not result in an improvement above a certain level, is dropped. The process is continued by successively refitting reduced models until further removal of variables does not result in any improvement. In the end, this means that only those variables that do contribute to the model are kept.

3.5.2 Forward selection

This method means adding a set of variables to a model one at a time in an order which corresponds to their ability to improve the model. At each step, each of the variables not already included in the model is tested for inclusion. The variable whose inclusion results in the greatest improvement is then added, provided that the improvement is significant or above a certain level. The process is ended when the addition of more variables from the set to the model does not improve it any further.

3.5.3 Optimization

Feature model optimization was done independently for each treebank. The reason for this was not only that linguistic features vary between treebanks, but also that the notation can vary between different treebanks; a feature model may be considered treebank-specific.

In these experiments, backward selection was first started on the baseline feature model. Each feature in turn was removed from the base model, and the accuracy (measured in LAS) of these reduced models was compared. The model displaying the most improvement when compared to the base feature model was kept, and the rest of the models discarded. Then the process was repeated using this new feature model as a temporary base model. When no further improvement in accuracy could be seen, the feature model was considered sufficiently optimized.

In addition to the 14 features in the baseline feature model, another 23 features were obtained by comparing other optimized feature models to the baseline feature model. It should be noted that two of the features available for forward selection try to extract LEMMA, which is not present in the Danish treebank, so these features are by definition not relevant to Danish. Nevertheless, all 23 features were tested for inclusion in the feature models through forward selection.

When no further improvement in accuracy could be seen, the feature model was considered sufficiently optimized. All treebanks were split in ten equal parts and ten-fold cross validation performed in order to minimize the risk of overfitting the model.

Tests were also performed with MaltParser using LibSVM with the quadratic kernel. For each treebank, MaltParser was run in learning and parsing mode using first the baseline feature model, then the language-specific, LIBLINEAR-optimized feature model. The purpose of these additional tests was to see if the improvement in accuracy obtained when using the optimized feature model with only LIBLINEAR would also be noticeable when using LibSVM.

For all tests, accuracy was measured as LAS using MaltEval.

4 Results and discussion

4.1 Initial tests

The results of the initial tests with LIBLINEAR indicated that solver type 4 (MCSVM by Crammer & Singer) would give the best accuracy when combined with a fairly low value assigned to the C parameter (0.1–0.2 for all treebanks). Table 4.1 shows the best LAS obtained for each treebank as well as the associated value of C.

Treebank	Best LAS	C value
Slovene	64.7%	0.2
Danish	80.2%	0.1
Arabic	63.9%	0.1
Turkish	70.6%	0.2

Table 4.1: Initial LIBLINEAR tests (LAS)

Treebank	LIBLINEAR	LibSVM (linear)
Slovene	64.7%	64.7%
Danish	80.2%	* 81.9%
Arabic	63.9%	* 65.9%
Turkish	70.6%	71.3%

Table 4.2: Initial tests, LIBLINEAR vs LibSVM (linear) (LAS)

These results were then compared to those obtained when using the LibSVM learner with the linear kernel. As seen in table 4.2, the results are similar, but LAS is slightly lower when using the LIBLINEAR learner. The differences in LAS are only statistically significant for Danish and Arabic, the two treebanks which also display the largest difference in LAS between the two classifiers, at 1.7 and 2.0 percentage points, respectively.

Classification with LIBLINEAR and LibSVM with the linear kernel should produce approximately equal results; the only difference to be expected is that LibSVM with the linear kernel should be slower (Hsu et al., 2008). The fact that LibSVM with the linear kernel outperforms LIBLINEAR for two of the four treebanks is thus notable, but may possibly be explained by the difference in the algorithms. The Danish treebank is larger than any of the other treebanks used in these experiments. This should not affect the accuracy of LIBLINEAR

negatively, since it is explicitly stated that LIBLINEAR is to be used for large linear classification.

In order to determine whether the difference in accuracy between LibSVM’s linear kernel and LIBLINEAR is acceptable, or indeed if it is specific to Danish and Arabic, further tests are required, preferably on even larger treebanks, since LIBLINEAR’s advantages should lie in classifying large amounts of data.

4.2 Feature model optimization

Table 4.3 shows the final feature models as optimized using MaltParser with the LIBLINEAR learner. The 14 features above the horizontal line are equivalent to the baseline feature model, whereas the features below that line are those additional features found to increase the accuracy for one or more of the tested treebanks. Of the 23 features tested for inclusion in each of the feature models, only 10 occur in any of the final optimized models, 7 features occurring in more than one optimized model.

Feature	Slovene	Danish	Arabic	Turkish
POSTAG, Stack[0]	+	+	+	+
POSTAG, Input[0]	+	+	+	+
POSTAG, Input[1]	+	+	+	+
POSTAG, Input[2]	+	+	+	+
POSTAG, Input[3]	+	+	+	+
POSTAG, Stack[1]	+	+	+	+
DEPREL, Stack[0]	+	+	+	
DEPREL, ldep(Stack[0])		+	+	+
DEPREL, rdep(Stack[0])	+	+	+	+
DEPREL, ldep(Input[0])		+		+
FORM, Stack[0]	+	+	+	+
FORM, Input[0]	+	+	+	+
FORM, Input[1]	+	+	+	+
FORM, head(Stack[0])	+	+	+	+
POSTAG, Stack[2]			+	
LEMMA, Stack[0]			+	
LEMMA, Input[0]	+		+	+
Split(FEATS, Stack[0]),\	+	+	+	+
Split(FEATS, Input[0]),\	+	+	+	+
POSTAG, succ(Stack[0])		+		+
POSTAG, pred(Input[0])	+	+	+	
Split(FEATS, Input[2]),\	+			
CPOSTAG, Stack[2]				
FORM, rdep(Stack[0])	+	+		
FORM, pred(Input[0])			+	

Table 4.3: Optimized feature models

When comparing the optimized feature models for each treebank, there are some similarities to be observed. Including the baseline features, a total of 13

features are common for all the optimized feature models. This is a fairly large share of the total number of features in each optimized feature model; these vary from 17 (Turkish) to 20 (Arabic).

4.2.1 Accuracy

The results after optimizing the feature model using LIBLINEAR are shown in table 4.4. Improvement in accuracy ranges from slightly more than 3 percentage points for Arabic to 5.5 percentage points for Turkish. An explanation for this may be that the features available here for backward and forward selection were simply not as well-suited for Arabic as they were for, in particular, Turkish.

The improvement in accuracy obtained when optimizing the feature models with the LIBLINEAR learner is clearly paralleled by that obtained when using LibSVM with the quadratic kernel and the optimized feature models. These results are shown in table 4.5. Again, the greatest improvement is seen for Turkish (6.1 percentage points) and Slovene (5.5 percentage points). These improvements are greater than when using LIBLINEAR. For Arabic and Danish, the improvement is roughly 2.9 percentage points and 1.6 percentage points, respectively, which is actually less than the improvement when using LIBLINEAR.

Feature model	Slovene	Danish	Arabic	Turkish
Baseline	62.1%	75.8%	60.9%	66.7%
Optimized	65.8%	79.8%	64.1%	72.2%

Table 4.4: LAS before and after optimization using LIBLINEAR

Feature model	Slovene	Danish	Arabic	Turkish
Baseline	62.9%	81.2%	64.1%	68.0%
Optimized	68.4%	82.8%	67.0%	74.1%

Table 4.5: LAS before and after optimization using LibSVM (quadratic kernel)

4.2.2 Learning and parsing times

	Slovene	Danish	Arabic	Turkish
Learning	7:33	49:54	21:02	69:10
Parsing	2:50	4:32	3:29	5:24

Table 4.6: Learning and parsing times in minutes, seconds; LibSVM (quadratic kernel)

Table 4.7 shows the learning and parsing times for LIBLINEAR, using the baseline feature model. Table 4.6 shows the learning and parsing times for LibSVM using the same baseline feature model. The computer used for these

	Slovene	Danish	Arabic	Turkish
Learning	0:12	0:34	0:19	0:22
Parsing	0:04	0:05	0:05	0:06

Table 4.7: Learning and parsing times in minutes, seconds; LIBLINEAR (MCSVM by Crammer & Singer)

experiments was equipped with an Intel Core2 Duo processor at 2,66 GHz, 4 Gb memory, and running a Linux system.

Obviously, using LIBLINEAR results in much smaller learning and parsing times. For example, when applying the learner to the largest of the four treebanks, Danish, the learning time when using LibSVM is about a hundred times greater than when using LIBLINEAR.

In fact, for all four treebanks, when using LIBLINEAR it never takes more than a minute to train the parser; parsing is then usually accomplished in a matter of seconds. This difference in learning and parsing times should be even more notable when performing tests on larger treebanks.

5 Conclusions

In this thesis, the possibility of using a machine learning package for linear classification (LIBLINEAR) in combination with the MaltParser system for data-driven dependency parsing has been explored.

Several related questions were posed in the initial chapter of this thesis. How would using only linear classification, as opposed to the present default MaltParser classifier, LibSVM, affect the speed and the accuracy of the system? If, as may be assumed, LIBLINEAR is faster than LibSVM, would it be possible to use LIBLINEAR in the sometimes very slow process of feature model optimization?

The initial experiments, aimed at reaching a good value for the C parameter as well as choosing what LIBLINEAR solver type to use, showed (somewhat surprisingly) that while the results measured in accuracy are comparable to those when using LibSVM with the linear kernel, they are not quite identical. However, learning and parsing with LIBLINEAR is certainly a much faster process.

The subsequent set of experiments concerned feature model optimization, and the results can be discussed from two different aspects.

First, with regards to accuracy: while the exact improvement varied between different treebanks, an improved accuracy when using only LIBLINEAR is also seen when using the LIBLINEAR-optimized feature model for learning and parsing with LibSVM and its quadratic kernel.

Second, and possibly even more interesting, with regards to learning and parsing times: LIBLINEAR is many times faster than LibSVM when learning and parsing the same amounts of data. The difference is more prominent during the learning phase than during the parsing phase, and becomes greater with larger amounts of treebank data. The time it takes to train a model on, for example, the Danish dependency treebank, is only a few minutes when using LIBLINEAR.

To conclude, using the LIBLINEAR classifier for feature model optimization is faster than LibSVM, thereby allowing for optimization of feature models on large treebanks without having to split the training data. It therefore seems reasonable to explore further the use of the LIBLINEAR classifier in order to take advantage of its speed.

Bibliography

- A. Abeillé, editor. *Treebanks: Building and Using Parsed Corpora*, volume 20 of *Text, Speech and Language Technology*. Kluwer Academic Publishers, Dordrecht, 2003.
- N. B. Atalay, K. Oflazer, B. Say, and Informatics Inst. The Annotation Process in the Turkish Treebank. In *Proc. of the 4th Intern. Workshop on Linguistically Interpreteted Corpora (LINC, 2003)*.
- A. Böhmová, J. Hajič, E. Hajičová, and B. Hladká. The PDT: a 3-level annotation scenario. In Abeillé (2003), chapter 7.
- S. Buchholz and E. Marsi. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the CoNLL-X*, pages 149–164, 2006.
- M. A. Covington. A Fundamental Algorithm for Dependency Parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102, 2000.
- K. Crammer and Y. Singer. On the Learnability and Design of Output Codes for Multiclass Problems. In *In Proceedings of the Thirteenth Annual Conference on Computational Learning Theory*, pages 35–46, 2000.
- S. Džeroski, T. Erjavec, N. Ledinek, P. Pajas, Z. Žabokrtský, and A. Žele. Towards a Slovene Dependency Treebank. In *Proceedings of Fifth International Conference on Language Resources and Evaluation*, 2006.
- R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research*, 9(4): 1871–1874, 2008.
- J. Hall. *MaltParser - An Architecture for Inductive Labeled Dependency Parsing*. PhD thesis, 2006.
- C. Hsu, C. Chang, and C. Lin. A Practical Guide to Support Vector Classification, October 2008. URL <http://www.csie.ntu.edu.tw/~cjlin>.
- M. Kay. Guides and oracles for linear-time parsing. In *Proc. 6th Internat. Workshop on Parsing Technologies, IWPT 2000*, pages 6–9, 2000.
- M. Kromann. The Danish dependency treebank and the underlying linguistic theory. In *Proc. of the Second Workshop on Treebanks and Linguistic Theories (TLT)*, 2003.

- R. McDonald and J. Nivre. Characterizing the Errors of Data-Driven Dependency Parsing Models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 122–131, 2007.
- J. Nilsson and J. Nivre. MaltEval: An Evaluation and Visualization Tool for Dependency Parsing. In *Proceedings of the Sixth International Language Resources and Evaluation*. LREC, May 2008.
- J. Nivre. Algorithms for Deterministic Incremental Dependency Parsing. *Comput. Linguist.*, 34(4):513–553, 2008.
- J. Nivre. *Inductive Dependency Parsing*. Springer, 2006.
- J. Nivre. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 49–56, 2003.
- J. Nivre and R. McDonald. Integrating Graph-Based and Transition-Based Dependency Parsers. In *Proceedings of ACL-08: HLT*, pages 950–958, 2008.
- J. Nivre, J. Hall, S. Kübler, R. McDonald, J. Nilsson, S. Riedel, and D. Yuret. The CoNLL 2007 Shared Task on Dependency Parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 915–932, 2007.
- K. Oflazer, B. Say, D. Z. Hakkani-Tür, and G. Tür. Building A Turkish Treebank. In Abeillé (2003), chapter 15.
- J. Hajič and O. Smrž, P. Zemánek, J. Šnaidauf, and E. Beška. Prague Arabic dependency treebank: Development in data and tools. In *Proceedings of the NEMLAR International Conference on Arabic Language Resources and Tools*, pages 110–117, 2004.
- O. Smrž, J. Šnaidauf, and P. Zemánek. Prague Dependency Treebank for Arabic: Multi-Level Annotation of Arabic Corpus. In *Proceedings of the International Symposium on Processing of Arabic*, pages 147–155, 2002.
- V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.