Analysis of Different Oracle Function Created With Machine Learning for Dependency Parsing

Kjell Winblad

October 5, 2010
Master's Thesis in Computing Science, 30 credits
Supervisor at IT-UU: Olle Gällmo
External Supervisors at STP-UU: Joakim Nivre, Marco Kuhlmann
Examiner: Anders Jansson

Uppsala University
Department of Information Technology
Box 337
SE-751 05 Uppsala
SWEDEN

Abstract

Dependency parsing has got increased attention the recent years for parsing natural language because of the robustness of the technology and that it gives good parsing accuracy. The reason for this is that dependency grammars has some properties which makes it possible to create parsing systems that can be trained by applying machine learning techniques on a data set that contains syntax trees for sentences without any need for a formal grammar.

People that have worked on improving dependency parsing systems have suffered from the long training times caused by the big data sets that are in use. That problem can potentially be solved without loosing much in parsing accuracy by dividing training data by a feature before the training with Support Vector Machines. In this thesis report it has been shown that the division of the training data and a decision tree method even can increase the accuracy of the parsing system and at the same time decrease the training time substantially. The dependency parsing system MaltParser has also been extended with an option for using a decision tree as machine learning method.

Analys av olika maskininlärningsmetoder för skapande av orakelfunktioner till Dependensparsning

Sammanfattning

Dependensparsning har fått en ökad popularitet de senaste åren för parsning av naturligt språk på grund av att tekniken är robust och att den ger ger bra parsningsresultat. Detta beror på att dependensgrammatik har några egenskaper som gör att det går att sätta upp parsnings-system som kan tränas med maskininlärningstekniker på en datamängd som innehåller syntaxträd för meningar utan att en formell grammatik behövs.

Människor som har arbetat med att förbättra resultatet för dependensparsnings-system har lidit av de långa träningstiderna som orsakas av de stora datamängderna som användas. Det problemet kan potentiellt lösas utan att förlora mycket i kvalitet på parsningsresultatet genom att dela upp träningsmängderna före träningen med *Support Vector Machine*er. I den här rapporten har det visats att uppdelningen av träningsdata och en beslutsträdsmetod till och med kan förbättra parsningsresultatet samtidigt som träningstiden förkortas betydligt. Dependensparsning-systemet MaltParser har också utökats med ett alternativ för att använda beslutsträd som maskininlärningsmetod.

Contents

Chapter 1

Introduction

To automatically generate syntax trees for sentences in a text, is of interest for many applications. It can be used in for example automatic translation systems and semantic analysis. The supervisors of this thesis work are doing research for the Computational Linguistics Group at Uppsala University about data driven dependency parsing of natural language. As a part of their research they are developing a data-driven dependency parsing system called MaltParser [NHN06][1]. During their work on MaltParser the supervisors found a phenomenon that was unexpected. The phenomenon occurred in a part of the system called the oracle function. The oracle function is a central part of the system that decides which of a number of possible parsing steps the system will take given the current state of the parser. One of the best method known so far for creating the oracle function is a machine learning method called Support Vector Machines (SVMs) [YM03]. A SVM is a classifier that given a vector of features can predict a given output. To do that it needs to go through a training phase in which it is fed with already classified training examples. The training phase is very memory and computationally expensive and to speed it up it is possible to divide the input data in a reproduce-able way and then train many smaller SVMs that can be combined to produce the final classifier. This usually produce a resulting classifier with worse accuracy. However, when a linear SVM was used together with division of the training data, the Computational Linguistics Group found that the result was better with division than without. This is interesting to investigate since the training and testing time can be reduced significantly if the division technique could be refined to give similar accuracy as the non-linear SVM. Furthermore, it could give insight into the classification problem which could lead to other improvements.

This thesis project contains an investigation about the unexpected result described above. The project also contains a theoretical study as well as experimentation with the aim to explain the unexpected result. Finally, a practical part has been done where a implementation of a new training module for MaltParser has been implemented.

Jokim Nivre at the Computational Linguistics department at Uppsala University has been the main supervisor for the project. He together with Marco Kuhlman and me have had regular meetings during the project. During this meetings we have discussed the results of the work and experiments since the previous meeting and based on that discussion we have decided what would be interesting to investigate next.

The Computational Linguistics group at Uppsala University has machine learning and machine translation as two of it's major research areas. The group is one of the best of it's

[1] MaltParser is an open source software package that can be downloaded from: http://maltparser.org

kind in the world and has organized many international conferences and workshops. Most recently they organized Annual Meeting of the Association for Computational Linguistics (ACL)[2] in the summer of 2010.

The rest of this chapter describes the problems that are dealt with in this thesis work and the goals of the thesis as they were stated in the beginning of the work. Chapter 2 which is called *Background* will explain the technologies that are needed to be able to understand the results of this thesis, namely Dependence Parsing and SVMs. Chapter 3 which is called *Hypotheses and Methodology* explains the hypotheses that are tested by the experiments and describes the software that have been used as well as the data sets. Chapter 4 which is called *Experiments* goes through the experiments performed and discusses the aim of them as well as the results. Chapter 5 which is called *MaltParser Plugin* describes the implementation and usage of the new plugin created for MaltParser. Finally, chapter 6 called *Conclusions* discusses the achievements of the work as well as it's limitations and possible further work.

1.1 Problem Description

This chapter describes the initial tasks as they were formulated in the beginning of the project. It also describes the goals of the project and why these goals were desirable to accomplish. How the goals are met is described chapter 6.

1.1.1 Problem Statement

The aim of the thesis work is to study dependence parsing and in particular the oracle function used to determine the next step in the parsing procedure. Different machine learning methods for creating this oracle function will be created and analyzed. Some results indicate that when a linear SVM is used the performance of the classifier can increase if the training set is divided to create several classifiers that are concatenated to create the final classifier. Division in this context means that every instance of the training data is mapped to one out of several partitions depending on the value of a feature. One aim of this project is to find an explanation for this result. If it turns out that the experiments and studies of machine learning methods shows that it could be useful to implement a feature for that in the parsing system MaltParser, it will also be a part of this project to do such an implementation, if the time limit permits.

1.1.2 Goals

The goals of this project can be summarized in the following list:

1. To do an investigation to find out more in detail than what previously have been done, how division effects the performance of the oracle function. Performance in this case refers to training time (the time it takes to train the classifier), testing time (how fast the classifier is when classifying instances) and the accuracy (how well the resulting oracle function perform inside the dependency parsing system in terms of measures as e.g percentage of correct labels compared to a correct syntax tree). This is interesting because it is known that division in some cases have positive effect on the accuracy and always have positive effect on training and testing time, but it is not very clear in which situations it has positive effect on accuracy. More insight into this can lead to

[2] Homepage for Association for Computational Linguistics: http://www.aclweb.org

new methods for implementing the oracle function which may produce faster parsers with acceptable or possible even better accuracy than has been obtained so far.

2. To do an investigation about why division in some situations has positive effect. The purpose of this is to understand what causes the positive effect of division. This may lead to new ideas about how to improve the accuracy of the classifier as well as new insight into the characters of the classification problem.

3. If the investigations described above shows that it could be useful to implement an alternative oracle functions into MaltParser this could also be done within the scope of the thesis. Even if the results does not show that a oracle function with better accuracy than the best so far which is a non-linear SVM with the Kernel Trick, it could be useful with a classifier that works nearly as well and have much better training and testing time. The difference in training time for a linear SVM and a non-linear SVM with the Kernel Trick is big. A shorter training time could be of help when parsing methods are tested. A typical training time for a non-linear SVM on a dependence parsing problem can be about a week and the linear version of the classifier can be trained within a few hours.

Chapter 2

Background

This chapters explains the two subjects Dependency Parsing and Support Vector Machines (SVMs) that are necessary to have an understanding of in order to understand the rest of the chapters in this report. Dependency Parsing is studied in the research fields computational linguistics and linguistics. SVMs are studied in the research field Machine Learning.

A Dependency Parser is a system that parse sentences from a natural language into tree structures that belong to a Dependency Grammar and to do that it often makes use of the machine learning methods. One of the machine learning methods that have shown the best results is SVMs. How all these terms fit together will be explained in the rest of the sections in this chapter.

2.1 Dependency Grammar

A Dependency Grammar like other grammatical frameworks describe the syntactic structure of sentences. Dependency Grammar differ from other grammatical frameworks in that the syntax tree is described as a directed graph where the labeled edges represents dependences between words. The following gives an example of such a structure:

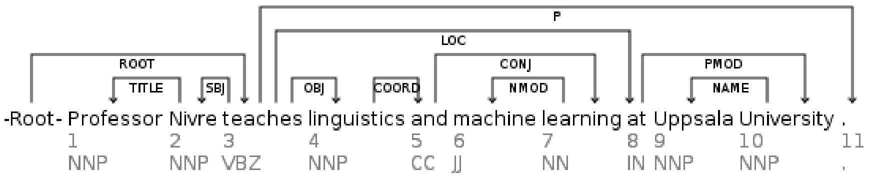
P
LOC
ROOT
CONJ
PMOD
TITLE
SBJ
OBJ
COORD
NMOD
NAME
-Root- Professor Nivre teaches linguistics and machine learning at Uppsala University .
1 2 3 4 5 6 7 8 9 10 11
NNP NNP VBZ NNP CC JJ NN IN NNP NNP .

Figure 2.1: The figure shows the dependency grammar structure for a sentence. E.g the node University has a edge labeled PMOD NAME to the word Uppsala, which describe a dependency relation between University and Uppsala.[1]

Most other grammar frameworks represents the structure of sentences with graphs where the words are leaf nodes which can be connected by relations and the relations can be connected by other relations to form a connected tree. It is possible to build hundreds of

[1] The figure is created by the open source tool ”What’s Wrong With My NLP?” that can be found at http://whatswrong.googlecode.com

different dependency trees for a normal sentences, but just a few of then will make sense semantically to a human. Due to the complexity of natural languages and the ambiguity of words, it is a very hard problem to automatically construct a dependency tree for an arbitrary sentence. One of the main reason for the popularity of dependency grammars in the linguistic community is that there exist simple algorithms that can generate dependency trees in linear time with fairly good result. Such an algorithm will be described in the next section. The introduction section about dependency grammar in the book Dependency Parsing by Nivre, Kübler and McDonald [KMN09] is recommended for information about the origin of dependency grammars etc.

2.2 Dependency Parsing

Dependency Parsing is the process of creating Dependency Grammar graphs from sentences. There exist grammar based dependency parsing systems as well as data driven dependency parsing system. The grammar based systems have a formalized grammar that is used to generate dependency trees for sentences and data driven systems make use of some machine learning approach to predict the dependency trees for sentences by making use of a large set of example predictions created by humans. Many system have both a grammar based component and a machine learning component. A grammar based system can make use of data driven approach to generate the formalized grammar. Some grammar based system generate many candidate dependency graphs and use a machine learning technique to select one of them.

In the rest of this section a data driven dependency parsing technique called transition-based parsing will be described. The parsing technique used in the experiments conducted in this thesis work is very similar to the one described here which is a bit simpler to give an an understanding of the method without going into unnecessary details. The system used in the experiments is called Nivre Arc-eager [Niv08].

A transition-based parsing system contains three components, namely a state, a set of rules that can transform a state to an other state and finally an oracle function that given a state selects a rule to apply. The components differ in different variants of transition-based systems. The system described in this section is a summary of the example system described in the chapter called Transition-Based Parsing in the book Dependency Parsing by Nivre, Kübler and McDonald [KMN09] and can be called the basic system.

The basic system has a state that consists of the three components presented in the following list:

– A set of labeled arcs from one word to an other word. The set is empty in the initial state.

– A stack containing words that are partly parsed. The stack only contains the artificial word ROOT in the initial state.

– A buffer containing words still to be processed. The first element in the buffer can be said to be under processing since an arc can be created between it and the top word in the stack and there is a rule that replaces the first word in the buffer with the first word on the stack. In the initial state the buffer contains the sentence to be parsed with the first word of the sentence as the first position in the buffer and the second word in the sentence as the second word in the buffer etc. The state in which the buffer is empty defines the end of the parsing.

The following list describes the rules used in the basic system:

LEFT-ARC Pop the first word from the stack W_1 and add the arc $(W_2, label, W_1)$ from the first word in the buffer W_2 to W_1 with an arbitrary label to the set of arcs. A precondition for this rule is that W_1 is not the special word ROOT. This is to prevent any other word to be dependent on it.

RIGHT-ARC Pop the first word from the stack W_1, replace the first word in the buffer W_2 with W_1 and add the arc $(W_1, label, W_2)$ to the set of arcs.

SHIFT Remove the fist word in the buffer from the buffer and push it to the top of the stack.

The parsing algorithm is very simple. It works by applying the rules until the buffer is empty. Then if the dependency tree that can be constructed by the arcs in the arc set is not connected or if words are missing from the sentences a tree containing all words is constructed by attaching words to the special ROOT word. Both the basic system and the Nivre Arc-eager system used in the experiments have been proven be both sound and complete, which means that parsing will always result in a dependency tree and all possible projective trees can be constructed by the rules [Niv08].

The selection of which rule to apply in a given state needs to be done by an oracle that knows the path to a correctly parsed tree. The oracle can be approximated by any machine learning method. To be able to use a standard machine learning classifier the state needs to be transformed to a list of numerical features. Which features that are most useful for classification depends on which language should be parsed and the selection of features are often done by people with a lot of domain specific knowledge. One of the most successful machine learning method that have been used for oracle approximation is Support Vector Machines with the Kernel Trick. Most of the experiments carried out during the thesis work has been done with a Support Vector Machine without the kernel trick which has the advantage compared to the one with the kernel trick that the training and testing time is considerably faster. The general idea for how Support Vector Machines work will be explained in the next section. It has been proven that both the basic system and Nivre arc-eager can parse sentences with the time complexity $O(N)$ given the assumption that the oracle function approximation run in constant time [Niv08].

2.2.1 Measuring the Accuracy of Dependency Parsing Systems

There exit some measurements used to evaluate dependency parsing systems. The label attachment score is a commonly used measurement that is also used in the experiments presented in chapter 4. The label attachment score counts the number of words in the parsed sentences that have got the correct head attached to it with the correct label. The head of a word is the word that is depending on it. In the system presented in section 2.2 the word in the root of the tree will have the special ROOT word as head and the special ROOT word is not included when the label attachment score is calculated.

Other measures that are commonly used is unlabeled attachment score which is the same as label attachment score but without any check of the label and the exact match measure that counts the number of sentences that exactly match the reference sentence.

2.3 Support Vector Machines

Support Vector Machines (SVMs) is a machine learning technique for classification. The basic linear SVM can only separate classification instances that can belong to one of two classes that are linearly separable. Through extensions of the basic concept it is possible to use variants of the basic SVM to classify nonlinearly separable data into many classes. [BGV92]

Because SVMs is such an complicated topic involving advanced mathematical concepts, only the most fundamental idea behind it and the concepts necessary to understand the results of the thesis project will be explained here. Chapter 5.5 in the book Introduction to Data Mining by Tan and Steinbach and Kumar [TSK05] is recommended to get a more in depth explanation of SVMs.

2.3.1 The Basic Support Vector Machine Concept

The idea behind SVMs is to find the hyperplane in the space of the classification instances that separate the classes with the maximum margin to the nearest instance. This is illustrated in figure 2.2 where the bold line is the hyperplane in 2-dimension space that separate the square class from the circle class with the maximum margin. The two parallel line illustrate where the borders are for which the margin should be maximized.

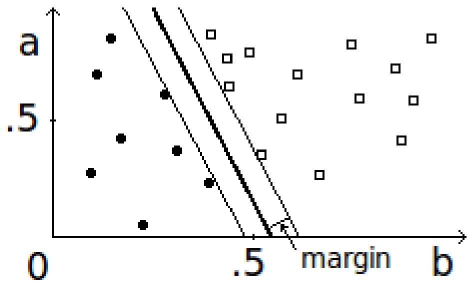

Figure 2.2: The figure that illustrate the maximum margin hyperplane that separate two classes in 2-dimensional space.

The training phase of a basic SVM is an optimization problem that tries to find the hyperplane with the maximum margin. In that process border points are found that are close to the border. These points are called support vectors and are used to calculate the maximum margin plane. In practice most training set is not linearly separable but the most basic SVM can be extended to support that by making a trade of between the distance from the separation hyperplane to the margin and the amount of misclassified training instances. [TSK05]

2.3.2 The Kernel Trick

The kernel trick is the name of a method to transform the space of the input space to a space with higher dimensionality. This is an effective technique to improve the accuracy of classifiers where the classification problem is not linearly separable. It has been showed that the accuracy of dependency parsing can be significantly improved if the the linear SVM used as oracle function in dependency parsing is replaced by a SVM that makes use of the

Kernel Trick. However, due to the higher dimensionality when the Kernel Trick is used both the training and testing time is much longer with the Kernel Trick. For an experimental comparison between the time complexity of systems that use the Kernel Trick and systems that do not use it, see section 4.1. SVMs with the Kernel Trick are sometimes referred to as nonlinear SVMs in this report and SVMs without the Kernel Trick are referred to as linear SVMs.

2.3.3 The Extension to Support Multiple Class Classification

The basic SVM only supports classification to one of two classes. There are many extensions that allows for SVMs to be used in multiple class classification problems. One popular approach which is both easy to implement and to understand is the one against rest extension. It works by creating one internal SVM for every class and train each internal SVM using one class for the class it is created for and an other class for the rest of the training instances. When an instance shall be classified all internal classifiers are applied to the instance and the resulting class is calculated by selecting the class that gets the highest score. The score can be calculated e.g. by giving one point to a class for every classification that supports the class. Which extension that gives best accuracy may differ for different problems [KSC$^+$08, TSK05].

The multiple class classification extention used for all experiments with linear SVMs in this thesis work is based on the method presented in the paper ”A Sequential Dual Method for Large Scale Multi-Class Linear SVMs” [KSC$^+$08].

2.4 Decision Trees

Decision Trees is an alternative machine learning method for classification to SVMs that also can be combined with SVMs or other machine learning methods to get improved result [SL91]. The basic principle behind decision trees is to divide a hard decision until it is only one decision or a high probability for one decision left. In the training phase of a decision tree a tree structure is build where the leaf nodes represents final decision and the other nodes represents divisions of the the original classification problem. As an example consider the dependency parsing system described in section 2.2. To make the example simple consider the very simple feature extraction model is used that takes out the type of the word on the top of the stack and the type of the first word in the buffer.

Top of stack	First in buffer	Rule
VERB	ADJE	**LEFT-ARC**
NOUN	ADJE	**RIGHT-ARC**
ADJE	NOUN	**RIGHT-ARC**
ADJE	VERB	**SHIFT**

Table 2.1: The table displays training examples for the decision tree example.

From the extremely small example training set presented in table 2.1 the decision tree in figure 2.3 could be constructed. As an example the last instance in table 2.1 would be classified by the decision tree by first going down from the root node in the tree along the branch marked s0(ADJE) and then follow the branch marked v0(VERB) where it would be classified to be of the **SHIFT** class because there it is a child node marked **SHIFT**

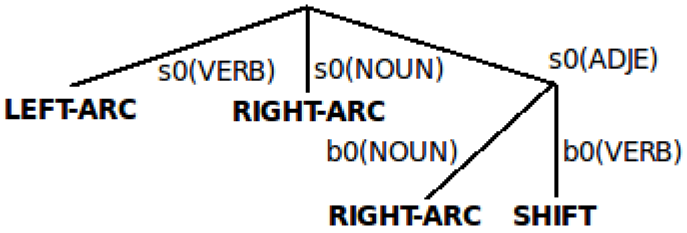

Figure 2.3: The figure shows an example of a decision tree where $s0(X)$ represent that the word on top of the stack has the word type X, and $b0(X)$ that the first word in the buffer has X as word type.

there. The example is too simple to be of any practical use. In real world applications it is necessary to make a trade off between training error and generalization error by having child nodes that have training instances belonging to more than one class. In such situations tested instances can be classified to be of the class that have the most training instances in that particular node. An other approach is to use an other machine learning technique as for example a SVM to train a sub classifier in that particular leaf node. This has been done in the experiments described in the section 4.5, 4.6 and 4.7.

There are many techniques to generate decision trees given a set of training instances. For more detailed information about that section 4.3 in the book Introduction to Data Mining by Tan and Steinbach and Kumar [TSK05] is recommended. The approach used in the experiments conducted in this thesis project is explained in section 4.5.

2.4.1 Gain Ratio

One problem when creating decision trees is how to know which feature to select to divide on first and second and so on. It desirable to have as few nodes in the tree as possible to reduce generalization error and at the same time to not have too few so the classifier have enough information to do a good classifications. Some measurements based on information theory have been developed to measure how much information is gained by dividing on a particular feature. Information in this context means how much better an arbitrary training instance can be classified after making use of the the knowledge gained from looking at one feature. A commonly used technique to calculate that is to look at the impurity of the the data nodes before and after splitting on a particular feature. A data node that contains only one class can be said to be totally pure and a data node that has the same number of instances from all classes is the most impure.

If only the information gain is used when deciding the split order for creating decision trees it tends to create trees with nodes that have many branches. This may not be optimal since the nodes get small early in the trees which can lead to generalization error and that features that could have lead to better prediction never get used. To get rid of these problems a method called Gain Ratio has been developed. Gain Ratio makes a trade of between trying to minimize the possible values of the feature selected and getting as good information gain as possible. This method is used in the experiments presented in section 4.6 and 4.7. Gain Ratio was developed by J.R. Quinlan [Qui93].

The Gain Ratio implementation used in the thesis work makes use of entropy as impurity measure. Entropy in information theory was introduced by Shannon and is a measure of the uncertainty of an unknown variable [Sha48]. It can be calculated as in equation 2.1 where

$p(i,t)$ is the fraction of instances belonging to class i in a training set t, c is the number of classes in the training set and $log_2(0)$ is defined to be 0. A more impure training set has a higher entropy than a less impure.

$$e(t) = -\sum_{i=1}^{c} p(i,t) log_2(p(i,t)) \tag{2.1}$$

The information gain is the difference between the impurity of a training set before and after splitting it with a certain feature. It can be calculated as in equation 2.2, where t is the parent training set, c is the number of sub training sets after splitting by the particular feature s, $t_1, t_2, ..., t_c$ are the training sets created by the split and $N(X)$ is the number of instances in training set X.

$$information\_gain(t,s) = e(t) - \sum_{i=1}^{c} \frac{N(t_i)}{N(t)} e(t_i) \tag{2.2}$$

The gain ratio measurement reduces the the value of information gain by dividing it with something that can be called Split Info as shown in equation 2.4. Split Info gets a higher value when there are more distinct values of a particular feature. Equation 2.3 shows how the Split Info is calculated, where t is the training set to be divided s is the split feature v is the total number of sub training sets created after splitting with s, $t_1, t_2, ..., t_c$ are the training sets created by the split and $N(X)$ is the number of instances in training set X.

$$split\_info(t,s) = -\sum_{i=1}^{v} \frac{N(t_i)}{N(t)} log_2(\frac{N(t_i)}{N(t)}) \tag{2.3}$$

$$gain\_ratio(t,s) = \frac{information\_gain(t,s)}{split\_info(t,s)} \tag{2.4}$$

For more detailed information about splitting strategies and alternative impurity measurements, section 4.3.4 in the book Introduction to Data Mining by Tan and Steinbach and Kumar [TSK05] is recommended.

2.5 Measuring the Accuracy of Machine Learning Methods with Cross-Validation

Cross-validation is a technique for measuring the accuracy of a machine learning method given only a training set. A training set is a set of classification instances with the correct classes associated with instances. The cross-validation procedure starts by dividing the training set in test partitions with equal number of instances in each. If the training set is divided in N test partitions the cross-validation is called N fold cross-validation. For every test partition created a bigger training partition is created by concatenating all other partitions. The cross-calculation accuracy is calculated by for every test partition first train a classifier with the particular machine learning method with the corresponding train partition and then test the resulting classifier on the test partition. The average of all the tests is said to be the cross-validation accuracy.

Chapter 3

Hypotheses and Methodology

This chapter describes which hypothesis for the experiments presented in chapter 4. It also describes the methods and tools as well as the data sets used to carry out the experiments.

3.1 Hypotheses

The following list contains descriptions of the hypotheses used when carry out the experiments described in chapter 4.

1. When a linear Support Vector Machine is used to create the oracle function for a dependency parsing system, the performance of the oracle function can become better if the training data is divided by a feature before the training. This hypothesis exists because of other previous undocumented experiments had indicated it.

2. The reason for the improvement described in *hypothesis 1* is that the classification problem for the whole input space is harder than the divided classification problem . In other words one can say that the SVM is not powerful enough to separate the classes in an optimal way, but technique where the SVMs get some help from an initial division is more powerful in that sense.

3. The smaller the partitions of the division becomes the more accurate the individual sub classifiers will become to some point when the accuracy will become worse because of lack of generalization. This is a well known principle in machine learning and this hypothesis was created to confirm that it applies to this particular problem as well. The hypothesis was created when experiments strongly supported *hypothesis 1 and 2* as a working hypothesis to improve the accuracy of the classifier even further.

4. There exists a way to automatically create a division strategy that can by itself decide when to stop the dividing to avoid lack of generalization. There exist a lot of intuitive knowledge about which features that are important for the classification in dependency parsing systems, but the knowledge may be language and system specific so a way automatically calculate a good division strategy can be of practically usefulness as well as theoretical interest.

3.2 Methods

The experiments described in section 4.1 and 4.7 were carried out with MaltParser. The rest of the experiments test different variants of machine learning methods with the same type of data that could be used in a dependency parsing system, but without a dependency parsing system. The reasons for that were to make the experiments easier to implement and to eliminate as many factors as possible to make the results easier to evaluate.

All experiments required a lot of computer calculation time as well as main memory due to the size of the training sets used in the experiments. Therefore they where executed on UPPMAX computer center[1]. The experiments were carried out on computers with Intel 2.66GHz quad core E5430[2] and 16GB of main memory.

UNIX Shell scripts and small programs written in the programming language Scala were created to automatize the experiment executions. All scripts and configurations can know be found at the following location ”http://github.com/kjellwinblad/master-thesis-matrial”. Due to the long execution process of the experiments errors in the scripts showed to be very time consuming to fix. E.g. after the execution of an experiment that took several days to finish, it was found that wrong parameters had been used, which made it necessary to rerun the experiment again with adjusted parameters. For that reason the process of writing experiment execution scripts was adjusted during the work to include more testing before the real experiment was started. The more extensive testing increased the development time for the experiments greatly.

An other thing that was learned during the work was that automatization of as much as the execution as possible was of great help. In the beginning of the project many steps in the experiments were done ”by hand” because they were believed to just be necessary to be executed once. This type of thinking showed up to be wrong. As an example things could show up later in other experiments that would make it interesting to rerun earlier experiments with a small adjustment which would be very easy and fast to do if there already was an automatic way to run the experiment, but quite cumbersome otherwise.

The results of the experiments were documented in a single document where graphs and result tables etc were pasted in. The results were also analyzed and discussed with the supervisors, but not much was written down about the analysis. In retrospective it would have been much better to spend more time on the experiment just performed to write down a everything that could be of interest from the analysis, instead of writing down all analysis in the end of the project.

3.3 Tools

Many different software tools have been used during the thesis work. In this sections the most important tools will be presented together with a short description. Some of the tools are irrelevant for the results of the experiments but will be presented anyway because the lessons learned from them might be of interest for other reasons.

3.3.1 MaltParser

MaltParser is a data driven transition-based dependency parsing system. It is written in the Java programming language. The main developers are Johan Hall, Jens Nilsson and Joakim

[1] UPPMAX is computing center hosted at Uppsala University. More information about the center can be found at the address ”http://www.uppmax.uu.se/”.

[2] The experiments only utilized one of the cores.

Nivre at Växjö University and Uppsala University. It has been proved to be one of the best performing system of it's kind by getting the top score in the competition CoNLL Shared Task 2007 on Dependency Parsing. The system is very configurable which makes it possible to optimize it for different languages. E.g. the feature extraction model can be replaced by using a configuration file, there are a number of different transition rules to chose from and different machine learning methods can be used. The system is also written to be easy to extend by writing plugins to replace components such as the machine learning method.

The MaltParser system can be run in two different modes. The first mode is the training mode where the input is configurations for the machine learning technique to use, a feature extraction model, dependency parsing algorithm settings and training data consisting of sentences with corresponding dependency trees. The output of the training phase is a model used to build the oracle function approximation used to decide the next step during parsing. The second mode is called parsing mode which takes a model created in the training mode and sentences to parse. The output is sentences with corresponding trees in the same format as the training set. The MaltParser settings used in the experiments are described in appendix B.

To measure the accuracy of the parsed sentences an external tool named conll07.pl[3] has been used.

3.3.2 The Support Vector Machine Libraries LIBSVM and LIBLINEAR

LIBLINEAR and LIBSVM[4] are two SVM implementations that are integrated into MaltParser. Both LIBSVM and LIBLINEAR are licensed under open source licenses. The original versions of the libraries are written in C but there exist Java clones as well as interfaces to many other programming languages for both libraries. LIBLINEAR implements linear SVMs and LIBSVM implement support vector machines with the Kernel Trick [CL01, FCH$^+$08]

The experiments described in chapter 4 that use LIBLINAR configures it with the parameters **-s 4 -c 0.1** and the ones that use LIBSVM configures it with the parameters **-s 0 -t 1 -d 2 -g 0.2 -c 1.0 -r 0.4 -e 0.1**. The parameters have been chosen because they have given good results previously. See the documentation for respective library for information about what the parameters mean.

3.3.3 The Programming Language Used for the Experiments: Scala

The Scala programming language was used to create the programs used in many of the experiments presented in chapter 4. The Scala programming language[5] is developed by the research group LAMP at Swiss Federal Institute of Technology Lausanne (EPFL). It is a mix between a functional language and an object orientated language and is made to be scalable from very small programming projects to very big programming projects. The compiler compiles the Scala code to Java byte-code and Scala can use Java classes without any configuration.[OAC$^+$04]

[3] The measurement tool can be found in the dependency parsing wiki http://depparse.uvt.nl/depparse-wiki/SoftwarePage

[4] LIBSVM and LIBLINEAR can be downloaded from http://www.csie.ntu.edu.tw/ cjlin/liblinear/ and http://www.csie.ntu.edu.tw/ cjlin/libsvm/.

[5] The Scala programming language is an open source project and the software can be found at http://www.scala-lang.org.

Scala was chosen as the language to write the experiments in mainly because it seemed to be simple and efficient for doing small programs which the programs for the experiments would be. An other reason was the interest and curiosity of the author of this report. Because the author of the report had no previous experience of Scala before the project started it may not have been used in the best possible way, but the experience was that it was very fast to learn with previous experience of Java programming and some functional programming. It is also believed that the language suited the task of writing experiments very well since it is possible to write clear code without side effects in it.

As described in section 3.2 it was very important to get the programs that executed the experiment right before the main experiment was run. To accomplish this tests were written in a Unit Test framework called ScalaTest at the end of the project. To write such tests helped a lot for writing bug free code. To summarize it can be said that Scala suited the task very well but it is not recommended to start working with a new programming language when it is critical to finish a project fast.

3.4 Data Sets

The training data sets used in the experiments described in chapter 4 are in the CoNLL Shared Task[NRY07] data format which is based on the MaltTab format developed for MaltParser. The data sets (also called treebanks) consists of sentences where the words are annotated with properties such as word class and with corresponding dependency trees.

The treebanks come from the training sets provided by the the CoNLL shared task. The treebanks named *Swedish*, *Chinese* and *German* in table 3.1 are the same as the training sets provided by the CoNLL-2006 shared task[BM06]. The treebanks named *Czech*and *English* come from the CoNLL-2009 shared task[HCJ+09]. A name for each treebank used in the report together with information on the number of sentences and number of words contained in them are listed in table 3.1.

Name	**Sentences**	**Tokens**	**Source**
Swedish	11042	173466	Talbanken05[NNH06]
Chinese	56957	337159	Sinica treebank[CLC+03]
German	39216	625539	TIGER treebank[BDH+02]
Czech	38727	564446	Prague Dependency Treebank 2.0[HPH+06]
English	39279	848743	CoNLL-2009 shared task[HCJ+09]

Table 3.1: Table with all treebanks used in the experiments in chapter 4.

Chapter 4

Experiments

In the following sections, the experiments conducted in the this work are presented. The experiments are related to the hypothesis presented in section 3.1 and often refer to the different hypotheses in the *Results and Discussion* sections of the experiments. The experiments can be seen as dependent on each other because after an experiment was finished, the next experiment to be conducted was decided and that decision was lead by the results of the previous experiments. The experiments were conducted in the same order as they are presented in this chapter. For parameters used for LIBLINEAR and LIBSVM in the experiments see section 3.3.2. For description of the data sets used in the experiments see section 3.4.

4.1 Division of the Training Set With LIBLINEAR and LIBSVM

The aim of this experiment is to look at differences in training time, parsing time and parsing accuracy when MaltParser is configured to divide the training data on a particular feature or not to divide the training data and to use LIBLINEAR or LIBSVM as learning method. The experiment was done with three different languages to see if the results are the same independently of the language.

The following training methods were tested in the experiment:

- Linear SVM (LIBLINEAR) with division of the training set
- Linear SVM (LIBLINEAR) without division of the training set
- Nonlinear SVM (LIBSVM) with division of the training set
- Nonlinear SVM (LIBSVM) without division of the training set

When division was used the training data was divided by the feature representing `POSTAG` property of the first element in the buffer. A test set was picked out from the original data set containing 10% of instances. Eight different training set were created from the remaining training instances where one contained all training instances, the next one half and the third one contained one forth etc until the last one that contained $\frac{1}{128}$ of the original training instances. The MaltParser configuration used in the experiment is explained in appendix B.1.

4.1.1 Results and Discussion

		LIBLINEAR							
Size		**1/128**	**1/64**	**1/32**	**1/16**	**1/8**	**1/4**	**1/2**	**1**
Swedish Div	**TR**	0.01	0.01	0.02	0.03	0.06	0.18	0.40	0.78
	TE	0.01	0.01	0.01	0.01	0.01	0.02	0.04	0.05
	AC	62.08	65.98	69.16	72.82	75.85	78.27	79.87	81.87
Swedish	**TR**	0.01	0.01	0.02	0.05	0.12	0.27	0.55	1.08
	TE	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02
	AC	62.13	65.93	68.99	71.94	74.64	76.63	78.79	80.22
Chinese Div	**TR**	0.01	0.01	0.03	0.07	0.16	0.25	0.56	1.20
	TE	0.01	0.01	0.01	0.02	0.02	0.03	0.04	0.05
	AC	68.94	73.23	76.08	78.01	79.73	81.00	81.99	83.38
Chinese	**TR**	0.01	0.01	0.03	0.10	0.20	0.29	0.76	1.82
	TE	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.03
	AC	63.48	68.78	71.76	74.32	75.99	77.52	79.01	80.44
German Div	**TR**	0.02	0.03	0.07	0.16	0.35	0.69	1.36	2.51
	TE	0.02	0.02	0.03	0.03	0.05	0.06	0.10	0.15
	AC	66.50	70.02	73.89	75.94	77.40	79.21	80.54	82.24
German	**TR**	0.02	0.03	0.07	0.21	0.39	0.90	1.83	3.26
	TE	0.03	0.03	0.02	0.03	0.03	0.03	0.03	0.04
	AC	66.56	68.82	70.38	71.44	72.61	74.15	75.54	77.45

Table 4.1: Table containing the results for the tree languages Swedish, Chinese and German tested with a linear SVM with and without division. **TR** = *training time in hours*, **TE** = *testing time in hours*, **AC** = *Label Attachment Score*

The results presented in table 4.1 and table 4.2 show that the training and testing time is much greater for the tests using nonlinear SVM compared to the ones using linear SVM. It can also be seen that division gives better training time than without division. The training time for the linear SVM seems to grow close to linearly with the number of training instances. For the nonlinear SVM the training time seems to grow exponential with the number of training instances which explains why division has such positive effect on the training time for the nonlinear SVM. The testing time is greater with than without division for the linear SVM case. This is probably caused by the fact that the models are loaded from disk and more training instances causes more models which increases the overhead of loading from disk.

Diagrams displaying the Label Attachment Score (LAS) for the tests with the three languages can be seen in figure A.4, A.5 and A.6 that can be found in appendix A. From the diagrams it is easy to see that the positive effect of division seems to increase with the size of the training set. It is also possible to see that for the tests with the biggest training sets, the accuracy is very similar for with and without division when nonlinear SVM is used, but there is clear difference in accuracy for with and without division together with the linear SVM in advantage for division. That the difference in LAS for the linear SVM decreases when the size of the training set gets smaller suggests that division is more useful the bigger the training set is.

The difference in accuracy for with and without division together with linear SVM support *hypothesis 1*, which says that division on a particular feature can have positive effect

		LIBSVM							
Size		**1/128**	**1/64**	**1/32**	**1/16**	**1/8**	**1/4**	**1/2**	**1**
Swedish Div	**TR**	0.01	0.03	0.05	0.10	0.17	0.39	1.51	5.63
	TE	0.10	0.41	0.37	0.36	0.37	0.55	0.82	1.11
	AC	58.21	63.33	68.11	71.71	74.93	78.09	80.66	82.56
Swedish	**TR**	0.01	0.03	0.08	0.39	1.40	5.87	24.94	128.31
	TE	0.10	0.29	1.09	2.06	2.35	4.97	7.58	12.42
	AC	58.03	63.53	69.29	73.23	76.31	79.30	81.38	83.56
Chinese Div	**TR**	0.01	0.03	0.06	0.19	0.53	2.55	11.24	72.86
	TE	0.07	0.18	0.40	1.03	1.24	2.18	3.96	7.71
	AC	64.61	70.67	73.83	77.11	79.62	81.63	83.15	84.77
Chinese	**TR**	0.02	0.05	0.17	1.10	3.98	16.74	83.41	405.05
	TE	0.35	0.87	1.79	4.03	6.50	11.04	17.62	33.51
	AC	53.15	62.67	68.07	73.29	77.25	80.40	82.30	84.33
German Div	**TR**	0.05	0.08	0.11	0.21	1.18	3.70	17.81	77.91
	TE	1.24	1.37	0.75	0.80	1.65	2.32	4.13	7.59
	AC	68.64	72.70	75.45	77.33	79.35	81.07	83.05	84.82
German	**TR**	0.07	0.24	1.34	5.31	23.03	98.02	420.84	–
	TE	2.72	3.83	6.52	13.10	20.72	32.82	58.99	–
	AC	69.25	72.81	75.26	77.34	79.21	81.19	82.96	–

Table 4.2: Table containing the results for the tree languages Swedish, Chinese and German tested with a nonlinear SVM with and without division. The results for the *German* with the biggest training set is not included in the results because of too long calculation time. **TR** = *training time in hours*, **TE** = *testing time in hours*, **AC** = *Label Attachment Score*

on the accuracy. One explanation for the fact that the same difference does not exist for nonlinear SVM can be that it is more powerful than the linear SVM and hence can handle the harder undivided problem better than the linear classifier. Therefore, the nonlinear SVM can not get the same improvement from division. This support *hypothesis 2*, which states that the reason for improvement gained by division is that the division makes the classification problem easier. That less training data decreases the relative accuracy advantage for the linear SVM with division compared to without division supports *hypothesis 3*, which says that dividing to smaller partitions can lead to improvement of the accuracy until they are too small to have good generalization.

4.2 Accuracy of Partitions Created by Division

The experiment described in section 4.1 showed that an improvement of the accuracy can be accomplished if the training data is divided by the value of one feature. The selected feature is believed to be important which means that it's value is believed to have relatively high impact on which parsing step that should be taken next in a given parsing state. However, the importance of the value of the feature depends on the values of the other features so in some situations the importance of the chosen feature may be very low. The experiment described in this section was conducted to find out how the division effects the partitions it creates. In particular it was investigated if it is possible to see a pattern in the relationships between the sizes of the partitions and the accuracy of their predictions.

The training data for the the three languages *Swedish*, *Chinese* and *German* were divided by the same feature as in the experiment described in section 4.1. Every partition created by the division was trained with a linear SVM (LIBLINEAR) by 10 fold cross validation. The cross validation accuracy of every partition was recorded together with the size of the partitions. It is important to note that the cross validation accuracy is not the same as the Label Attachment Score (LAS) used when measuring the accuracy of parsing. However, they are closely related to each other because if the prediction of which parsing step should be taken in a given parsing state gets better, then it should result in a higher LAS because fewer errors will be made.

4.2.1 Results and Discussion

It is not possible to see any obvious correlation between the size of the partitions of the training data and it's cross validation accuracy. This is illustrated in the figure A.1, A.2 and A.3 that can be found in appendix A. It is noteworthy that some portions have as good as 99% accuracy and these partitions occur among both big partitions and among small ones. The median based box plots presented in figure 4.1 show how the accuracy vary among the partitions.

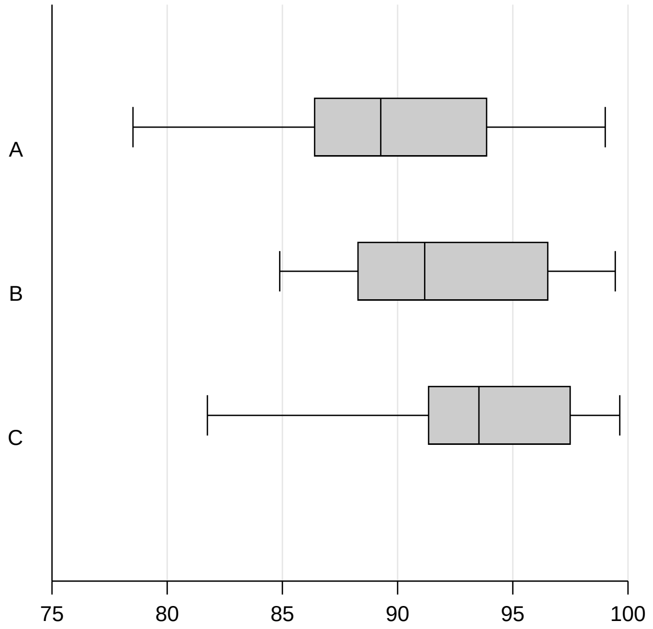

Figure 4.1: The diagram shows three median based box plots for the accuracy of the partitions created by division. Plot A is for Swedish, B is for Chinese and C is for German.

An explanation for the fact that some partitions get as good as 99% accuracy is that the division creates some partitions that are easy to create a linear classification model for. E.g. most instances of those partitions may belong to just a few classes that are easy to separate. That explanation support *hypothesis 2*, which says that the reason for the improvement gained by the division is that the division creates easier classification problems than all data together.

The accuracy of some partitions are worse then the accuracy of the classifier created by training all training instances together. That may lead to the hypothesis that the division is not optimal for the whole data set which will be explored in the experiment described in the next section.

4.3 An Other Division of the Worst Performing Partitions After the First Division

The hypothesis tested in this experiment is if the accuracy of the worst partitions created by division are bad because the feature selected for division was not relevant for the instances in these partitions. The experiment described in section 4.2 showed that the division that was made created some partitions with good accuracy and some with worse. The span of the accuracy of the partitions is big, which gives a reason to believe that the division was not the best for all partitions and that an other division or no division could be better for the worst performing ones.

All partitions that had worse than the weighted average accuracy for the partitions in the experiment described in section 4.2 were concatenated to a new chunk of training data. That new chunk of training data was then trained by cross validation and a linear classifier (LIBLINEAR) everything at once and after division with the feature representing the `POSTAG` property of the first element on the stack (feature 2). The feature used to divide the training data in the experiment described in section 4.2 represents the `POSTAG` property of the first element in the buffer (feature 1). The same was done for the partitions with better than average accuracy.

4.3.1 Results and Discussion

The results from the execution of the experiment is presented in table 4.3.

	Worse Than Average				**Better Than Average**				**Everything**		
Language	**Size**	**Feat. 1**	**Feat. 2**	**No div.**	**Size**	**Feat. 1**	**Feat. 2**	**No div**	**Feat. 1**	**Feat. 2**	**No div.**
Swedish	0.52	86.90	86.25	86.82	0.48	95.72	95.59	95.72	91.15	90.75	90.92
Chinese	0.64	89.50	89.39	89.57	0.36	96.75	96.43	96.61	92.09	91.72	92.04
German	0.39	88.10	87.91	85.84	0.61	95.54	95.41	94.93	92.68	92.52	91.04

Table 4.3: The table presents the results of the partitions performing worse and better separately when using feature 1 to divide the training instances. The columns named *Size* in the table represent the fraction of the total number of instances that where in the worst performing partitions and in the best performing partitions. The columns named *Feat. 1* represents the test case when the POSTAG property of the first element in the buffer was used to divide the instances. The columns named *Feat. 2* represents the test case when the POSTAG property of the first element in the stack was used to divide the instances. The columns named *No div.* represents the test case when all partitions were concatenated to one chunk that was trained with cross validation.

Division on feature 1 generally give better accuracy than feature 2. The partitions that had better than average accuracy when dividing on feature 1 seems to be about the same amount better than the rest even without division. Division with feature 2 gives worse result than no division at all for all languages except German.

The result does not indicate that the worst partitions after division with feature 1 were bad because the division had bad impact on them. Instead it seems like the division has good impact even on the partitions with worse than the weighted average for all language but Chinese were the accuracy is a little bit worse with division than without. The fact that dividing on an other feature did not make the partitions that had worse than average accuracy after division with feature 1 get much better or worse accuracy suggest that the training instances that they represent are harder to separate independently of which division feature is chosen to divide on. It is also possible that the feature 2 is very similar to feature 1. Another division feature could give an other result so more division features needs to be tested to be able to say something for certain.

4.4 Different Levels of Division

This experiment was created to see if an improvement of the accuracy could be made by dividing the the training data even more than what have been done in previous experiments. Seven different data sets were tested by 10 fold cross validation everything together, after division by the feature representing the $POSTTAG$ property of the first element in the buffer and after dividing the partitions created by the first division with the feature representing the $POSTTAG$ property of the first element in the stack. The average weighted accuracy was calculated from the cross validation results of the partitions created by division.

The *Swedish*, *Chinese* and *German* data sets are created by using the feature extraction model that can be found in appendix B. The feature extraction models used to create the data sets *Czech Stack Lazy*, *Czech Stack Projection*, *English Stack Lazy* and *English Stack Projection* can be found in appendix B.2.

4.4.1 Results and Discussion

	No Division	**1 Division**	**2 Divisions**
Swedish	90.9162	91.1496*	90.9787
Chinese	92.0441	92.0889	92.1678*
German	91.0386	92.6778	92.7075*
Czech Stack Lazy	89.9787	91.1751	91.8522*
Czech Stack Projection	89.7828	91.0158	91.6156*
English Stack Lazy	94.3736	94.7557	94.9537*
English Stack Projection	94.4000	94.7626	95.0080*
Average	*92.0479*	*92.7576*	*93.0013**

Table 4.4: The table shows weighted cross validation scores for different levels of division.

The results of the experiment is presented in table 4.4. An improvement is gained from one division compared to no division for all data sets and an even bigger improvement is gained from two divisions for all data sets except *Swedish*. The average improvement from one division to two divisions is about 0.24%. The average improvement from no division to 1 division is significantly larger namely 0.71%. *Swedish* is the smallest training set which can be an explanation of why the division had the least good effect on it. The partitions created by the second division on *Swedish* might be too small for the training to create general enough classifiers from them.

The results of this experiment supports *hypothesis 1*, which says that an improvement can be gained if the training data is divided before training. That Swedish got worse accuracy with two divisions than one division and that the improvement of the accuracy for all languages where bigger for the first division than the second support *hypothesis 3*, which states that the accuracy can be improved by division to a certain point. The results indicate that the point might be reached at one division for Swedish and that the point might be further away than two divisions for the other data sets.

So far it has just been assumed that the features that have been used for division in the fist and second division are good. It is possible that other features are better for the divisions and that might differ from language to language, because the same feature might have different impact on the grammatical structure of a sentence in different languages.

4.5 Decision Tree With Intuitive Division Order

The experiments described so far have indicated that the accuracy of the classifier can be improved by division. They also indicate that there is a limit where division starts to make the accuracy of the classifier worse instead of improving it. If that is true, the best classifier could be created by dividing the training data to that limit but not longer. This experiments aims to do that by creating a decision tree that has a creation strategy where it is tested for every division if the accuracy gets better or worse by doing cross validation.

A list of features ordered by intuitively importance was created. The intuition of the importance of the features is based on experiences gained by the supervisor of the thesis project Joakim Nivre during his research. The list is presented in table 4.5.

Feature Number	Element From	Element Property
1	Input[0]	POSTTAG
2	Stack[0]	POSTTAG
3	Input[1]	POSTTAG
4	Input[2]	POSTTAG
5	Input[3]	POSTTAG
6	Stack[1]	POSTTAG

Table 4.5: The table lists the intuitive division order used in the decision tree creation algorithm. Input[N] represents the n:th element on the buffer and Stack[n] represents the n:th value on the stack in the dependency parsing algorithm. E.g. Input[0] represents the first element in the buffer. POSTAG is the property used for all division features.

The decision tree was then created with the algorithm described in table 4.6. The algorithm is a recursive algorithm that returns an accuracy and with some small modifications a decision tree as result. The experiment was run with 10 fold cross validation and 1000 as minimum training set.

The *Swedish*, *Chinese* and *German* data sets are created by using the feature extraction model that can be found in appendix B. The feature extraction models used to create the data sets *Czech Stack Lazy*, *Czech Stack Projection*, *English Stack Lazy* and *English Stack Projection* can be found in appendix B.2.

Given:

- List of features to divide on L
- A training set T
- Minimum size of a training set created after division M

Algorithm:

1. Run cross validation on T and record the accuracy as A
2. If the size of T is less than M then return A as the result
3. If L is empty return A as the result
4. Divide T into several subsets so every distinct value of the first feature in L has it's own subset
5. Create an additional training set by concatenating all training sets created in 4 that has a size less than M
6. For all training sets created in step 4 and 5 except the ones concatenated because the size were less than M, run this algorithm again with L substituted with "L without the first element" and T substituted with the sub training set and collect the results
7. Calculate the weighted average accuracy WA from the results obtained in 6
8. If the weighted average accuracy WA is less than the accuracy without division A then return A as the result otherwise return WA as the result

Table 4.6: The table contains the decision tree algorithm used in the experiments.

4.5.1 Results and Discussion

The results of the experiment are summarized in table 4.7. Compared to the average accuracy obtained from two divisions in the experiment described in section 4.4 the decision tree gives an improvement of 0.11%. All training sets except *Chinese* had better accuracy with decision tree than the best obtained in the experiment described in section 4.4. The two first features in the feature division list used for creating the decision tree are the same as the two used for the experiment with two divisions in section 4.4. That *Chinese* got slightly worse result with the decision tree anyway can be explained by the fact that when division with one and two features have been used partitions that contains less than 1000 instances have been put in a separate training set called the other training set but with the decision tree there is one such other training set for every division. When looking at the structure of the decision trees created for the different training sets, it is possible to see that some nodes are divided more than others and that the maximum depth for the trees seems to increase with the size of the training set.

Hypothesis 3, which says that the classification accuracy of the problem can get improved by division to a certain point when it starts to get worse is strongly supported by the experiment. The experiment indicates that the accuracy can get improved by several division but also that there is a limit for the improvement. The experiment also supports *hypothesis 4*, which says that there is a way to automatically create a division to improve the accuracy.

	Intuitive	**Gain Ratio**
Swedish	91.1675∗	90.9472
Chinese	92.1179	92.1349∗
German	93.1317∗	92.9364
Czech Stack Lazy	91.8662	91.9473∗
Czech Stack Projection	91.6537	91.7707∗
English Stack Lazy	95.0205	95.1203∗
English Stack Projection	95.0770	95.1578∗
Average	*93.1178*	*93.0884*

Table 4.7: The accuracy for different language calculated in the decision tree experiment. The column named Intuitive represents the tree division with the intuitive division order and the column named Gain Ratio represent the tree division with division order calculated by Gain Ratio. Please see section 4.6 for an explanation of the Gain Ratio column.

The only thing that is not automatic in the training is the selection of the division features. If that also can be automatic is investigated in the experiment described in the next section.

4.6 Decision Tree With Devision Order Decided by Gain Ratio

The experiment described in section 4.5 indicated that combining a decision three with a linear SVM can improve the accuracy compared to a linear SVM without any division of the training data. It is likely that the improvements that could be gained is highly dependent on the division features used when creating the tree. One method often used to select division features when creating decision trees is called Gain Ratio. A description of the Gain Ratio measurement is provided in section 2.4.1. This experiment was set up to try the Gain Ratio as ordering measurement for the possible division features.

The experiment set up is exactly the same as in the experiment described in section 4.5 with the exception that the list of division features is not a qualified guess but sorted by the Gain Ratio measurement.

4.6.1 Results and Discussion

The results of the experiment are summarized in table 4.7. The average accuracy for the data sets trained with the the Gain Ratio calculated decision order is slightly worse 0.03% than when trained with the intuitive decision order, but still 0.08% better than when two divisions were used without any decision tree.

This experiment shows that we can get improvement with an algorithm that creates a division in a totally automatic way. This makes the decision tree method more interesting for practical use because no domain specific knowledge is required to use it.

4.7 Decision Tree in Malt Parser

All experiments described so far except the experiment described in section 4.1 have not been in a real dependency parsing setting. It is not obvious what effect a small improvement of the oracle function would have in a dependency parsing algorithm. The reason is that a misclassification by the oracle function does not automatically translate to an error in a dependency parsed sentence because errors in one parsing state can cause errors in later parsing states. The training data for the oracle function is also created from correctly parsed sentences and when errors have occurred in a previous parsing state it is less likely that the state or similar states is in the training data. Therefore, it is important to see what effect an improvement of the oracle function has in a real dependency parsing setting.

The decision tree creation methods described in section 4.5 and 4.6 are integrated into MaltParser. The implementation and usage of the MaltParser decision tree plugin is described in chapter 5. For all languages tested 10% of the instances of the original training set were removed and put in a testing set. For all tested languages 8 different sizes of the training set were tested. One contained all training instances, the next one half and the third one contained one forth etc. The set up is very similar to the set up used in the experiment described in section 4.1. Also the dependency parsing algorithm and feature extraction model are the same as used in the experiment described in section 4.1. As minimum size of a partition created by a division 50 was chosen. All partitions created by a particular division with a size less than 50 were concatenated to a new partition and if that new partition was smaller than 50 it was concatenated by the smallest partition created that is bigger than 50. To be able to compare the results with something the linear SVM with division experiments described in section 4.1 were run again but with 50 as minimum partition size.

4.7.1 Results and Discussion

The results of the experiment are summarized in table 4.8. Diagrams displaying the Label Attachment Score for the tests can be seen in figure A.7, A.8, A.9, A.10 and A.11, which can be found it appendix A. The results indicate what previous experiments also have indicated. The accuracy get better for most languages with a decision tree than division on just one feature. The intuitive division order have better accuracy for all data sets compared to the Gain Ratio generated division order. It is interesting to note that decision order decided by Gain Ratio seemed to perform better than the intuitive decision order for the more advanced feature extraction models used for *Czech* and *English* in the experiment described in section 4.6. The reason for that may be that the intuitive decision order is designed for a simple feature extraction model and that the Gain Ratio order can take advantage of the more advanced feature extraction model.

Tests were also performed with more advanced feature extraction models together with decision trees in MaltParser. The optimized feature extraction created models with more features than the standard, which causes them to consume more memory when loaded. This turned out to be a problem, since all of the models have the same numbers of support vectors the memory usage increase linearly with the number of models. The tests were run on computers with 16 GB of memory which was not enough to complete the experiments with all training data. For smaller subsets of the training data, the experiment indicated that the accuracy would get improved with tree division compared to devision on just one feature.

Liblinear Decion Tree in MaltParser

Size		**1/128**	**1/64**	**1/32**	**1/16**	**1/8**	**1/4**	**1/2**	**1**
Swedish Division	**TR**	0.01	0.02	0.03	0.06	0.10	0.16	0.28	0.35
	TE	0.01	0.01	0.01	0.01	0.02	0.03	0.04	0.05
	AC	59.63	63.88	68.14	72.30	75.97	78.57	80.99	82.28
Swedish Decision Tree	**TR**	0.02	0.03	0.03	0.12	0.23	0.48	0.85	1.59
	TE	0.01	0.01	0.01	0.02	0.02	0.03	0.04	0.06
	AC	59.48	65.48	70.25	72.14	75.87	78.50	80.97	82.18
Swedish Decision Tree Aut.	**TR**	0.01	0.02	0.03	0.09	0.21	0.51	0.88	1.65
	TE	0.01	0.01	0.01	0.01	0.01	0.03	0.06	0.10
	AC	59.86	65.65	70.30	71.55	76.34	77.67	80.28	82.06
Chinese Division	**TR**	0.02	0.05	0.08	0.22	0.41	0.64	0.98	1.67
	TE	0.02	0.02	0.02	0.05	0.05	0.07	0.13	0.22
	AC	54.92	61.89	67.24	71.15	75.23	78.37	80.57	82.54
Chinese Decision Tree	**TR**	0.02	0.03	0.06	0.12	0.28	0.61	1.04	3.73
	TE	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.25
	AC	63.64	69.48	73.02	76.10	77.78	79.33	80.75	82.56
Chinese Decision Tree Aut.	**TR**	0.02	0.05	0.11	0.38	0.39	0.77	1.45	2.81
	TE	0.01	0.01	0.02	0.02	0.02	0.03	0.04	0.07
	AC	62.52	68.51	72.91	75.77	77.35	79.31	80.76	81.54
German Division	**TR**	0.02	0.03	0.09	0.17	0.31	0.41	0.73	1.17
	TE	0.02	0.02	0.03	0.05	0.04	0.06	0.09	0.14
	AC	69.53	72.68	75.03	76.63	77.94	79.25	80.37	81.61
German Decision Tree	**TR**	0.08	0.09	0.22	0.54	1.22	2.16	4.86	12.33
	TE	0.03	0.02	0.03	0.06	0.07	0.13	0.31	1.18
	AC	69.57	72.72	74.99	76.75	78.48	80.18	81.45	82.98
German Decision Tree Aut.	**TR**	0.04	0.07	0.14	0.33	0.73	1.82	4.45	12.67
	TE	0.02	0.02	0.02	0.03	0.06	0.11	0.28	0.97
	AC	67.92	71.13	73.74	75.79	76.70	78.53	79.90	81.16
Czech Division	**TR**	0.02	0.03	0.04	0.09	0.22	0.26	0.44	0.71
	TE	0.02	0.02	0.03	0.03	0.04	0.04	0.06	0.08
	AC	53.91	56.41	59.30	61.51	63.63	65.64	67.20	69.10
Czech Decision Tree	**TR**	0.06	0.13	0.26	0.59	1.20	2.55	5.64	11.84
	TE	0.02	0.02	0.03	0.04	0.05	0.11	0.18	0.38
	AC	53.18	57.41	60.07	62.33	64.08	66.28	68.12	70.06
Czech Decision Tree Aut.	**TR**	0.03	0.08	0.17	0.40	0.77	1.70	3.68	12.76
	TE	0.02	0.02	0.03	0.04	0.05	0.10	0.20	0.53
	AC	53.93	57.32	60.06	62.32	64.05	66.04	67.89	69.56
English Division	**TR**	0.04	0.05	0.08	0.13	0.19	0.31	0.54	0.87
	TE	0.02	0.03	0.03	0.03	0.04	0.05	0.07	0.10
	AC	73.43	77.06	79.39	81.53	83.19	84.75	85.82	86.77
English Decision Tree	**TR**	0.09	0.15	0.26	0.43	0.92	1.88	4.06	11.44
	TE	0.03	0.03	0.03	0.04	0.05	0.09	0.19	0.35
	AC	73.27	76.96	79.50	81.57	83.31	85.01	86.31	87.32
English Decision Tree Aut.	**TR**	0.05	0.08	0.15	0.33	0.65	1.70	3.14	10.30
	TE	0.03	0.02	0.03	0.03	0.05	0.10	0.14	0.32
	AC	73.37	77.08	79.41	81.56	83.27	84.90	86.16	87.14

Table 4.8: Table containing the results for the decision tree algorithm with LIBLINEAR implemented in MaltParser. The decision tree creation algorithm is presented in section 4.5. **TR** = *training time in hours*, **TE** = *testing time in hours*, **AC** = *Label Attachment Score*

Chapter 5

MaltParser Plugin

As a part of the thesis work a plugin to MaltParser has been developed. The plugin adds a new machine learning method to MaltParser for creating the oracle function. The method is a combination of a decision tree and an additional machine learning method which is used to classify instances that belong to a certain leaf node. The decision tree is created in a recursive manner where a node become a leaf node if dividing the node more does not make the node get better accuracy. A detailed description of the algorithm used to create the decision tree can be found in section 4.5. The MaltParser plugin has been tested in the experiment described in section 4.7. This chapter contains an explanation of how the MaltParser plugin has been implemented as well as an explanation of how to use it.

5.1 Implementation

MaltParser is prepared for implementation of new machine learning methods. Before the implementation of the decision tree learning method there were tree main types to chose from, namely LIBLINEAR, LIBSVM and division strategy together with either LIBLINEAR or LIBSVM. The implemented decision tree plugin has many similarities to the division strategy method, which made it possible to use some functionality developed for the division strategy in the decision tree plugin.

5.2 Usage

As most configuration for MaltParser the decision tree alternative can be configured either by command line options or by options in a configuration file that can be passed to MaltParser. The options for the decision tree alternative are placed in the option group named `guide`. All options that are related to the decision tree alternative have names starting with `tree_`. The options have been documented in the MaltParser user guide. For an example of a decision tree configuration see appendix B.3.

The decision tree can be created either by manually configure a division order for the tree creation or by letting the program deduce a division order by calculating the Gain Ratio value for all possible features. As the experiments show it is not obvious which of the alternatives is best to use, since in the experiment the division order created by a person with a lot of domain knowledge worked better than the one created with Gain Ratio in many cases.

Besides the two different ways to select the division of the tree, there are some configuration options to put further constraints on the decision tree creation process. There are options for setting a minimum size of a leaf node in the tree, setting a minimum improvement limit for division of a node in the tree, the number of cross validation divisions to be made when evaluating a node and finally to force division on the root node to avoid cross validation on it.

Some tests have been made with different values on the parameters that showed that it is difficult to have any general principle of how they should be set. The minimum accuracy option is created to make it possible to reduce the risk of over-fitting the training data by making the tree more shallow. All tested values on that option have decreased the accuracy of the tree compared to when it is set to the default value 0 when 2 fold cross validation was used. The reason may be that the low number of cross validation divisions predicts low accuracy for training sets that are small, because the training sets in the cross validation become too small. If that reasoning is valid it is possible that higher number of cross-validation creates a tree with worse accuracy, but that the accuracy then can get improved by setting the minimum improvement option to a higher value. In that case it is smarter to use a low number of cross validation divisions which will result in a faster training.

For the option that decides the minimum size of a leaf node in the tree, some tests have been made with the value 50 and 1000. The tests do not indicate that one of the values are generally better than the other. It seems like it depends on the language and the size of the training set. The differences where also so small that it is difficult to tell if the difference only depends on particularities of the training data.

Chapter 6

Conclusions

The main goals of the projects have been described in section 1.1.2. In this chapter it will be discussed how well the goals are met in the project as well as interesting for future work.

Goal number 1 is to investigate how division of the training data effects the performance of the resulting dependency parsing system. This has been investigated in experimentally in the experiments described in chapter 4. The experiments show that when certain division features are used together with a linear SVM it can improve the accuracy of the resulting classifier in a significant way. The training and testing time are affected by division when a linear SVM is used as classifier, but the difference is so small that it is probably not important for most applications. However, the division approach could easily be scaled out so training of different partitions could be run in parallel which could potentially speed up the training time significantly. The experiment described in section 4.1 showed that division also could have a positive effect of the accuracy when nonlinear SVMs are used. To summarize the findings about how the accuracy is effected by division, one can say that it seems to depend on the training data and that the more training data there is the more positive effect can be gained by division. Not surprisingly division has a very positive effect on both training and testing time when division is used together with nonlinear SVM. The reason is that the training and classification time grows exponential with the number of training examples for nonlinear SVMs.

Goal number 2 is to investigate the theoretical reason for the effect of division. The hypothesis that has been discussed in this report and that many of the experiments in chapter 4 support, is that when division on a certain features are done it creates many smaller classification problems that the nonlinear SVM and linear SVM easier can separate into classes than the full classification problem. In other words, the SVM method together with division is more powerful than just the SVM in some cases. Intuitively it can be explained by the fact that a plane may not be able to separate all data, but if enough data is removed from the original data it will be able to separate it. The risk of dividing too much is that the classifier will over-fit the problem, which will lead to worse generalization and more errors when the classifier is tested. This is a well known principle in machine learning called over-training. An attempt to address this problem in form of a decision tree classifier is tested in the experiments described in section 4.5, 4.6 and 4.7 with fairly successful results.

Goal number 3 is to implement a new machine learning method in MaltParser based on the findings in the work. This has been done in form of a decision tree method where one of the libraries LIBSVM and LIBLINEAR is used as classifier in the leaf nodes. The

MaltParser plugin is described in chapter 5. A possible use case for the new method could be when parsing and training speed as well as good accuracy is important. The experiments have shown that division strategies have more positive effect on the accuracy if the training set is bigger. The training sets available for different languages will probably be bigger in the feature to increase accuracy and then the tree division strategy investigated in this project may become even more useful than it is today.

6.1 Limitations

One practical memory limitation was found in the experiment. To get a really good accuracy in the dependency parsing, optimized feature extraction parameters that extract a lot of features need to be used and even if the experiment indicate that an improvement could be gained even with such parameters the amount of RAM memory needed was a practical limitation for doing such experiments.

There were a lot of discussion during the project about additional experiments that just could not be done because of the time limitation of the projects. E.g. it would have been interesting to try other more advanced strategies for creating decision trees and compare them.

6.2 Future work

All the experiments that have been conducted during the thesis work in a real dependency parsing setting has used suboptimal feature extraction models and parameters. The reasons for that have been that it was desirable to keep the experiments fast to run and the memory usage within the limit that the available hardware provided. It would be interesting to see how the new decision tree based dependency parsing technique with optimized parameters perform compared to the state of the art dependency parsing systems. The amount of memory that the decision tree model needs when the dependency parsing system is in the parsing state is the biggest obstacle for archiving this. One approach to solve that is to swap out the SVM leaf models of the decision tree to disk when the memory gets full. This was tested during the project but was found to be unpractical, because the execution time become too long. A better solution would be to have the models loaded on several different computers. This could have the additional benefit of speeding up the parsing by parsing many sentences at the same time in parallel.

The speed of the training phase of the decision tree based model and the devision model could also be significantly improved by parallelization. It would be a quite simple job to implement this kind of parallelization by having model training servers that run on different computers and that could be asked to train a model on specific data set by the master system. The parsing parallelization could work in a similar way but instead of servers for training models the parsing servers can be responsible for loading models and serving classification requests.

References

[BDH^{+}02] S. Brants, S. Dipper, S. Hansen, W. Lezius, and G. Smith. The TIGER treebank. In *Proceedings of the workshop on treebanks and linguistic theories*, pages 24–41, 2002.

[BGV92] B.E. Boser, I.M. Guyon, and V.N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[BM06] Sabine Buchholz and Erwin Marsi. Conll-x shared task on multilingual dependency parsing. pages 149–164, 2006.

[CL01] C.C. Chang and C.J. Lin. LIBSVM: a library for support vector machines, 2001.

[CLC^{+}03] K.J. Chen, C. Luo, M. Chang, F. Chen, C. Chen, C. Huang, and Z.M. Gao. Sinica treebank: Design criteria, representational issues and implementation. *Abeillé (Abeillé, 2003)*, pages 231–248, 2003.

[FCH^{+}08] R.E. Fan, K.W. Chang, C.J. Hsieh, X.R. Wang, and C.J. Lin. LIBLINEAR: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

[HCJ^{+}09] Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia, Martí Lluís, Màrquez Adam, Meyers Joakim, and Nivre Sebastian Padó. The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages, 2009.

[HPH^{+}06] J. Hajic, J. Panevová, E. Hajicová, P. Sgall, P. Pajas, J. Štepánek, J. Havelka, M. Mikulová, Z. Zabokrtskỳ, and M.Š. Razımová. Prague Dependency Treebank 2.0. *LDC2006T01, ISBN*, pages 1–58563, 2006.

[KMN09] S. Kübler, R. McDonald, and J. Nivre. Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 1(1):1–127, 2009.

[KSC^{+}08] S.S. Keerthi, S. Sundararajan, K.W. Chang, C.J. Hsieh, and C.J. Lin. A sequential dual method for large scale multi-class linear SVMs. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 408–416. ACM, 2008.

[NHN06] J. Nivre, J. Hall, and J. Nilsson. MaltParser: A data-driven parser-generator for dependency parsing. In *Proceedings of LREC*, volume 6. Citeseer, 2006.

[Niv08] J. Nivre. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553, 2008.

[NNH06] J. Nivre, J. Nilsson, and J. Hall. Talbanken05: A Swedish treebank with phrase structure and dependency annotation. In *Proceedings of the fifth International Conference on Language Resources and Evaluation (LREC)*, pages 1392–1395. Citeseer, 2006.

[NRY07] 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*, pages 915–932, 2007.

[OAC$^+$04] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. *LAMP-EPFL*, 2004.

[Qui93] J.R. Quinlan. *C4. 5: programs for machine learning*. Morgan Kaufmann, 1993.

[Sha48] C.E Shannon. A mathematical theory of communication. Bell Syst. Tech. *Journal*, 27:1–65, 1948.

[SL91] S.R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–673, 1991.

[TSK05] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[YM03] H. Yamada and Y. Matsumoto. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3. Citeseer, 2003.

Appendix A

Experiment Diagrams

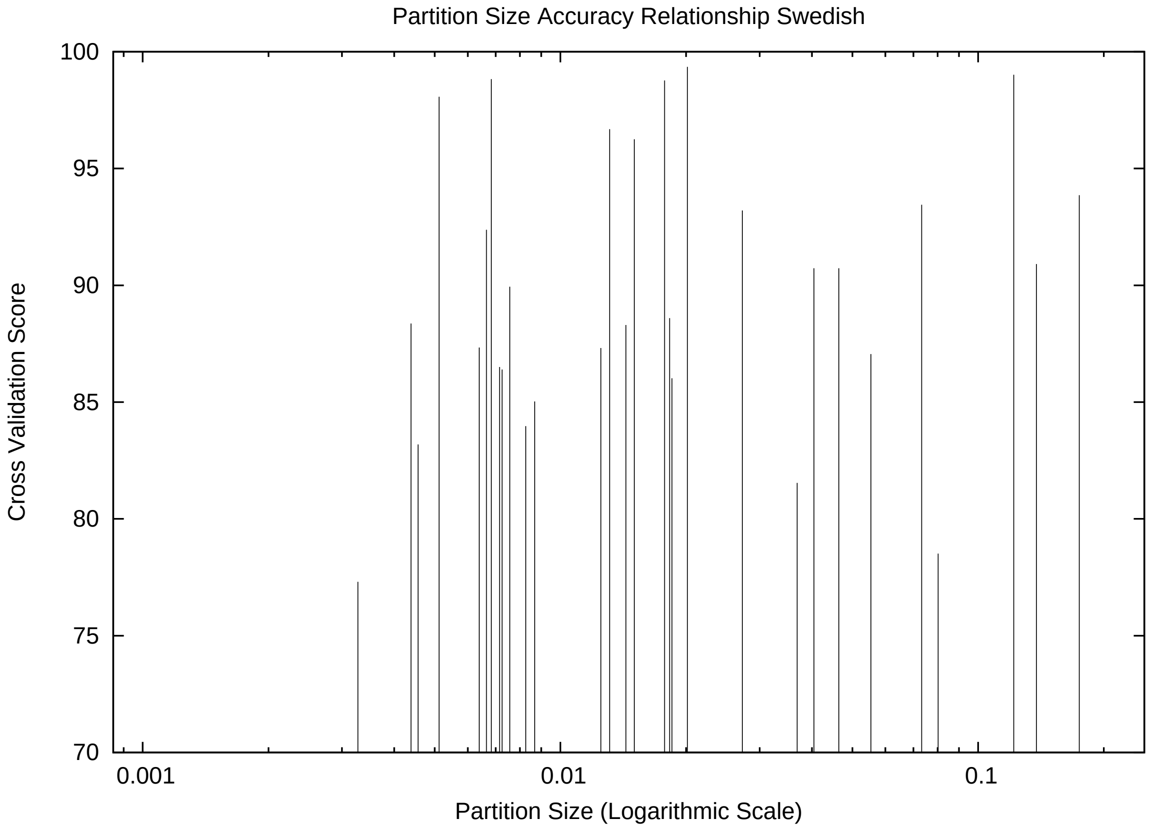

Figure A.1: The diagram illustrate that there is no clear pattern between size of partitions created when dividing the training data and the cross validation accuracy. Every spike in the diagram represents a partition created when dividing Swedish.

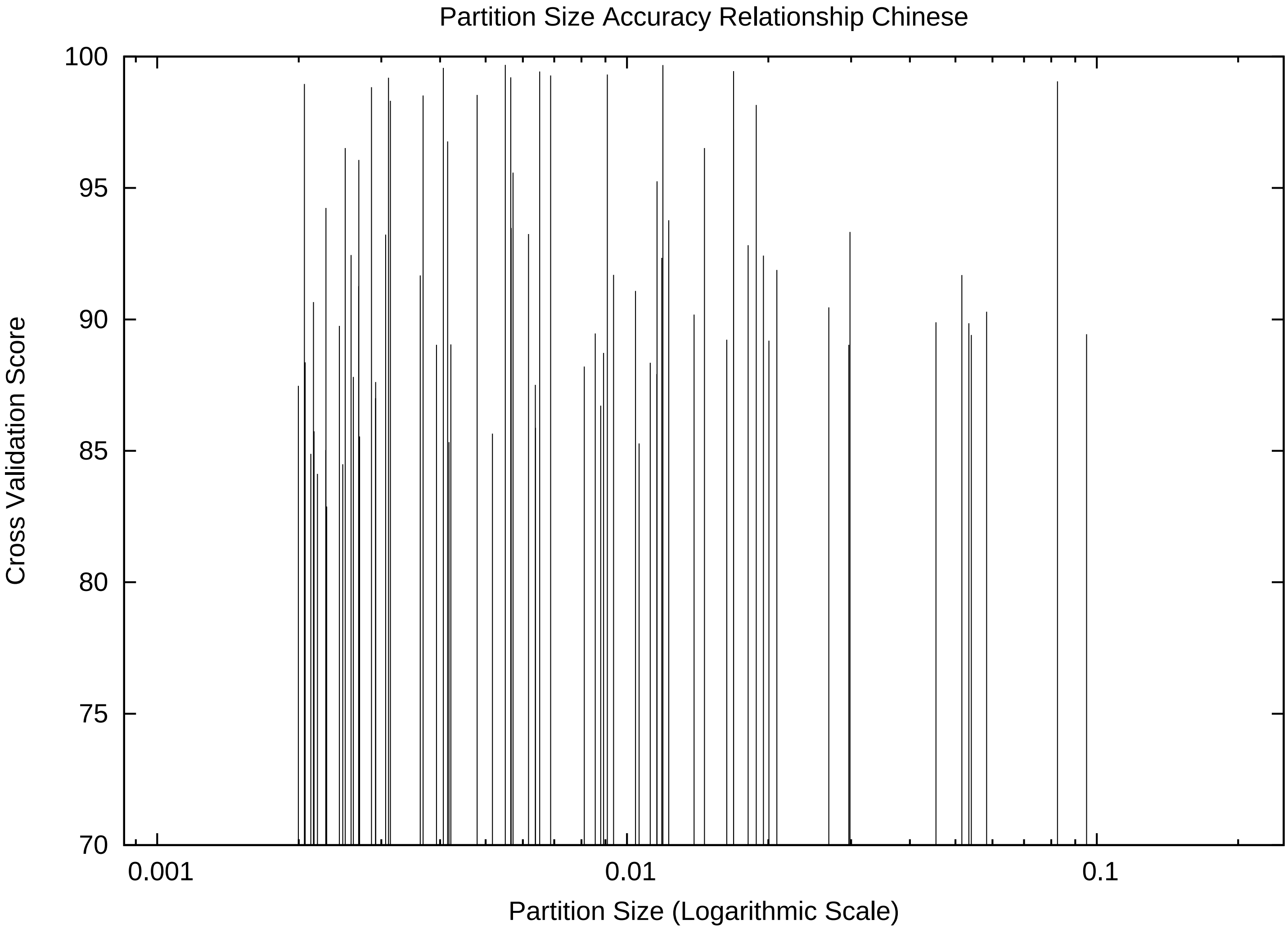

Figure A.2: The diagram illustrate that there is no clear pattern between size of partitions created when dividing the training data and the cross validation accuracy. Every spike in the diagram represents a partition created when dividing Chinese.

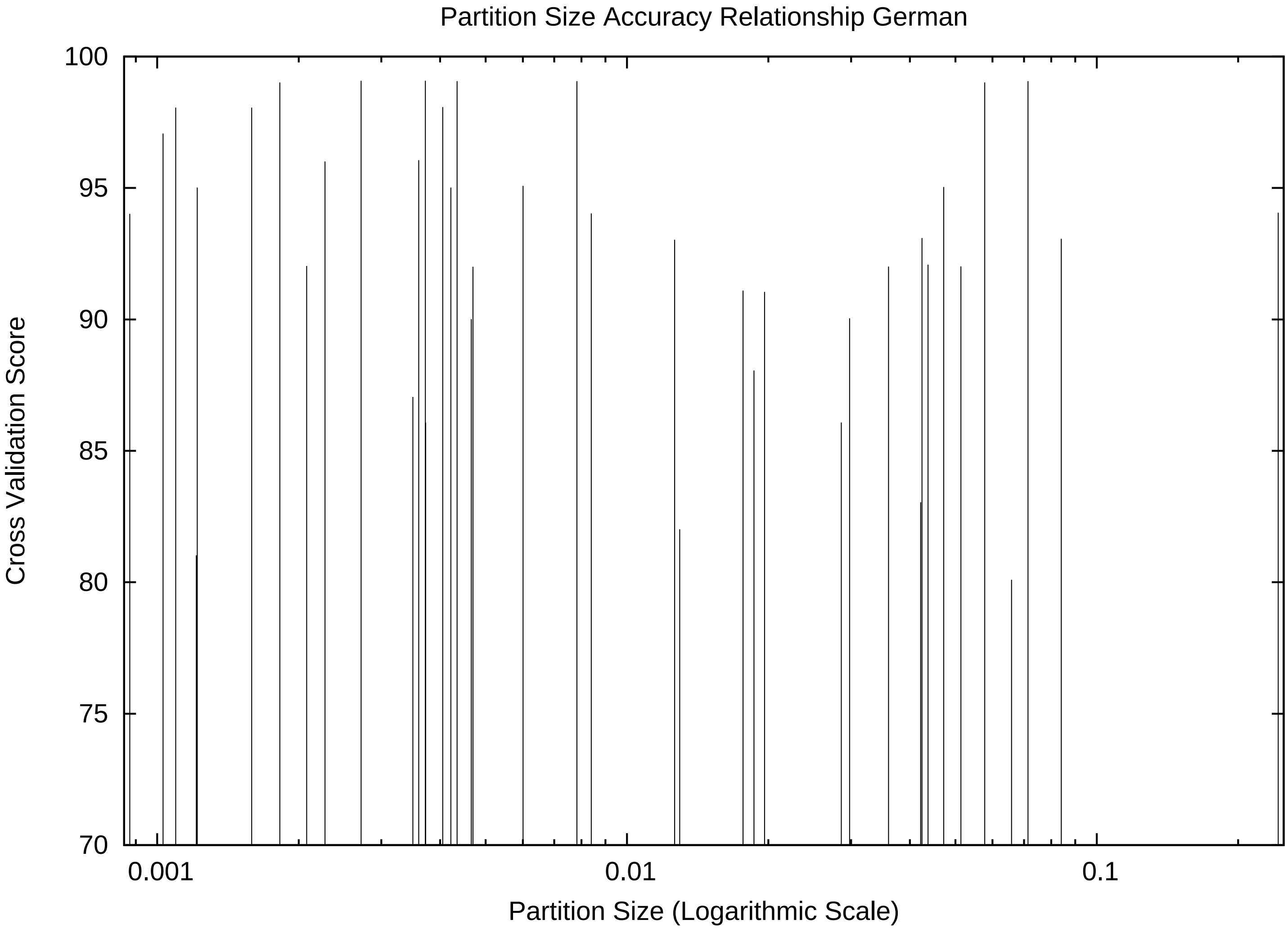

Figure A.3: The diagram illustrate that there is no clear pattern between size of partitions created when dividing the training data and the cross validation accuracy. Every spike in the diagram represents a partition created when dividing German.

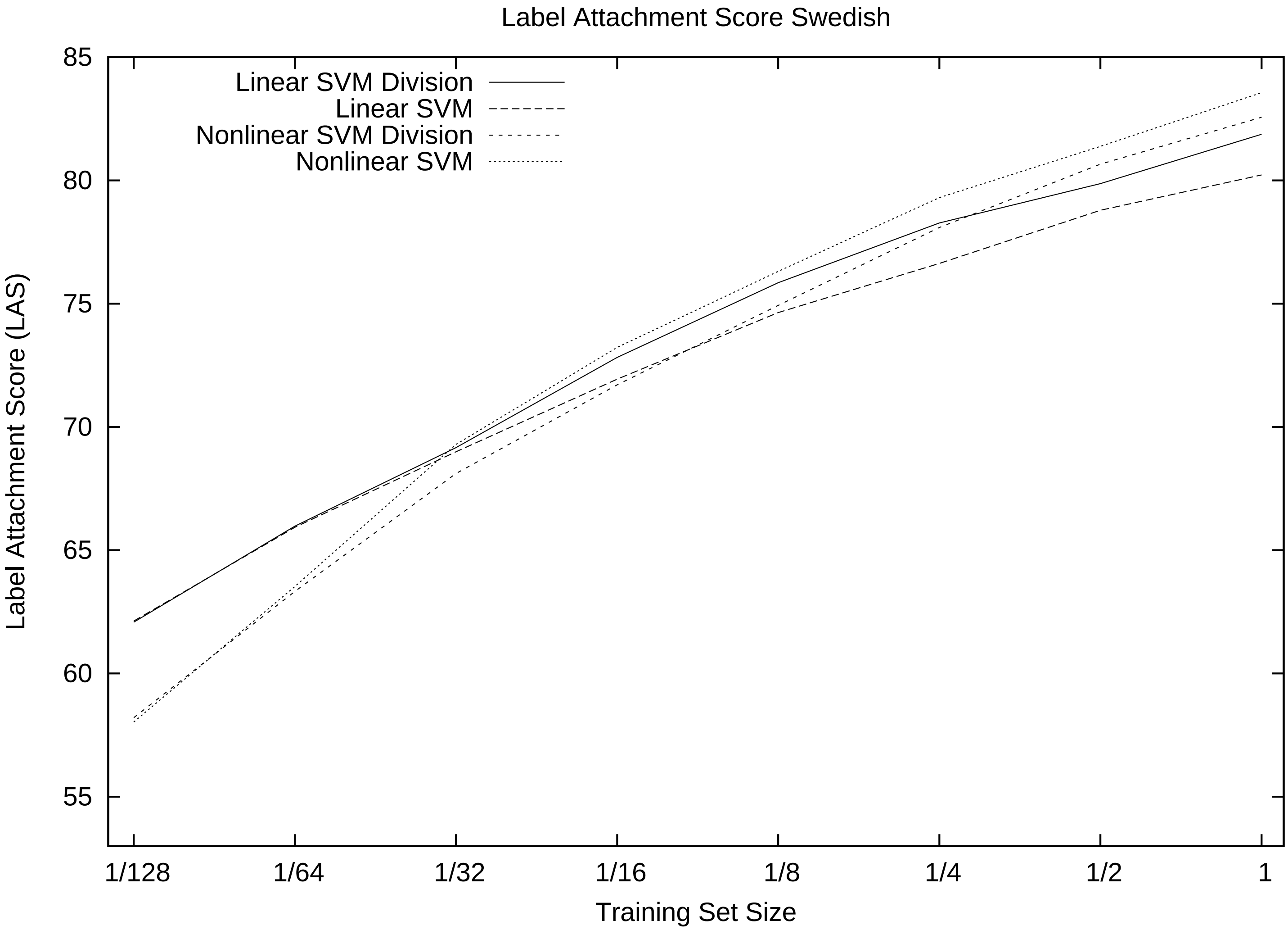

Figure A.4: The digram visualize the Label Attachment Score for all training set sizes created from the Swedish data set and tested in the experiment presented in section 4.1. The values used to create the diagram can be seen in table 4.1 and 4.2. Please, note that the x-axis in the diagram has logarithmic scale.

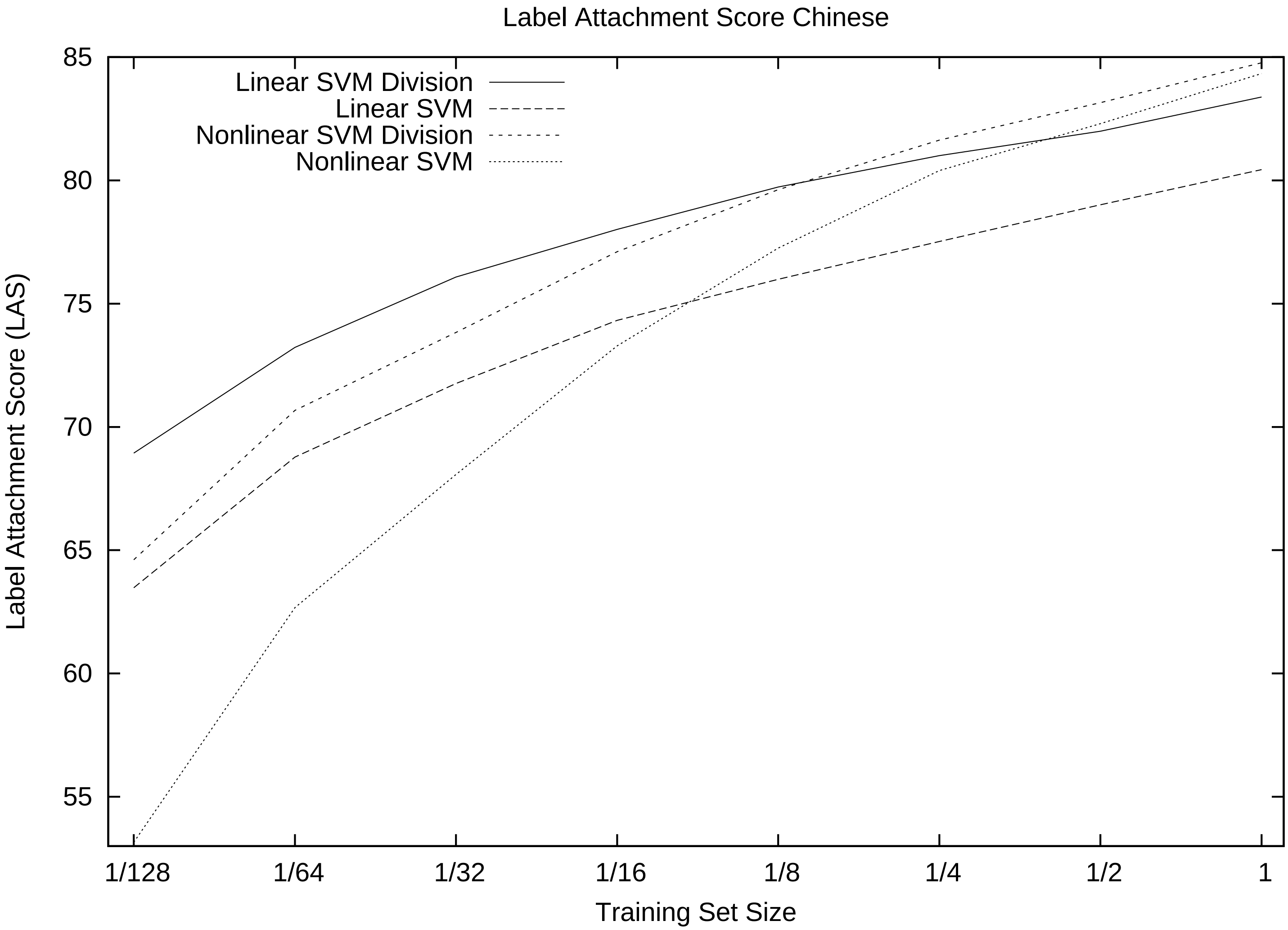

Figure A.5: The digram visualize the Label Attachment Score for all training set sizes created from the Chinese data set and tested in the experiment presented in section 4.1. The values used to create the diagram can be seen in table 4.1 and 4.2. Please, note that the x-axis in the diagram has logarithmic scale.

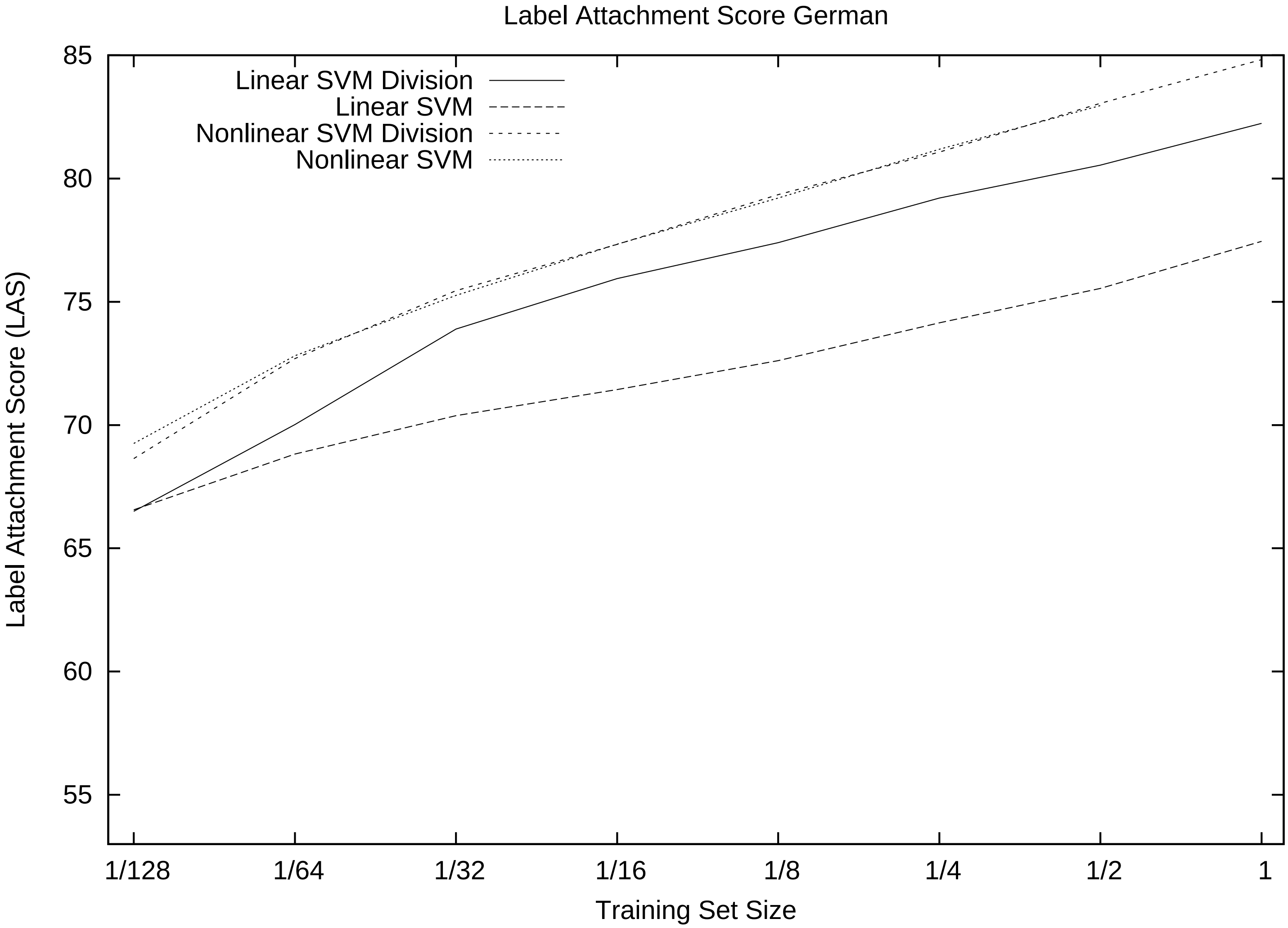

Figure A.6: The digram visualize the Label Attachment Score for all training set sizes created from the German data set and tested in the experiment presented in section 4.1. The values used to create the diagram can be seen in table 4.1 and 4.2. Please, note that the x-axis in the diagram has logarithmic scale.

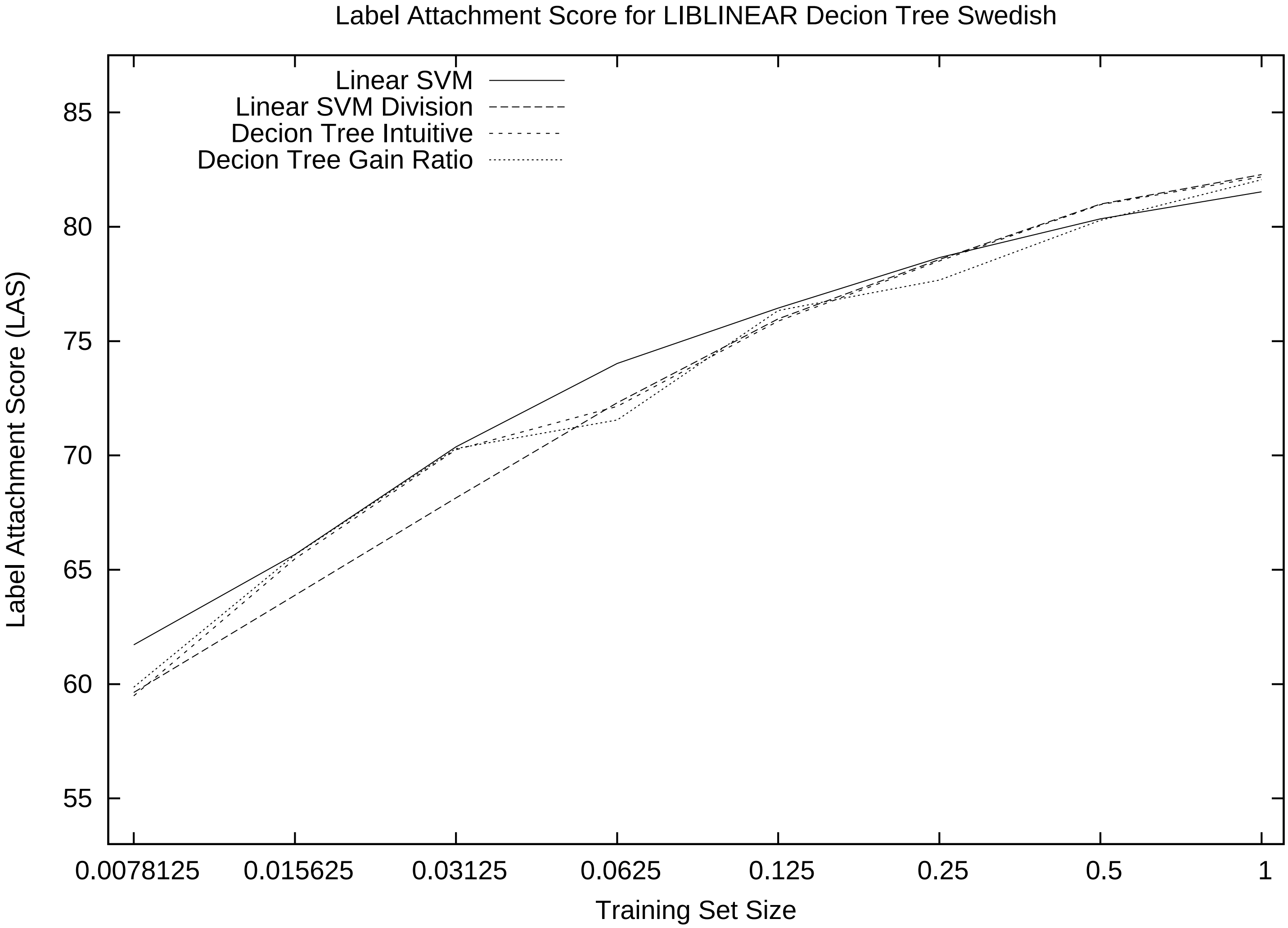

Figure A.7: The digram visualize the Label Attachment Score for all training set sizes created from the Swedish data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.1 and 4.8. Please, note that the x-axis in the diagram has logarithmic scale.

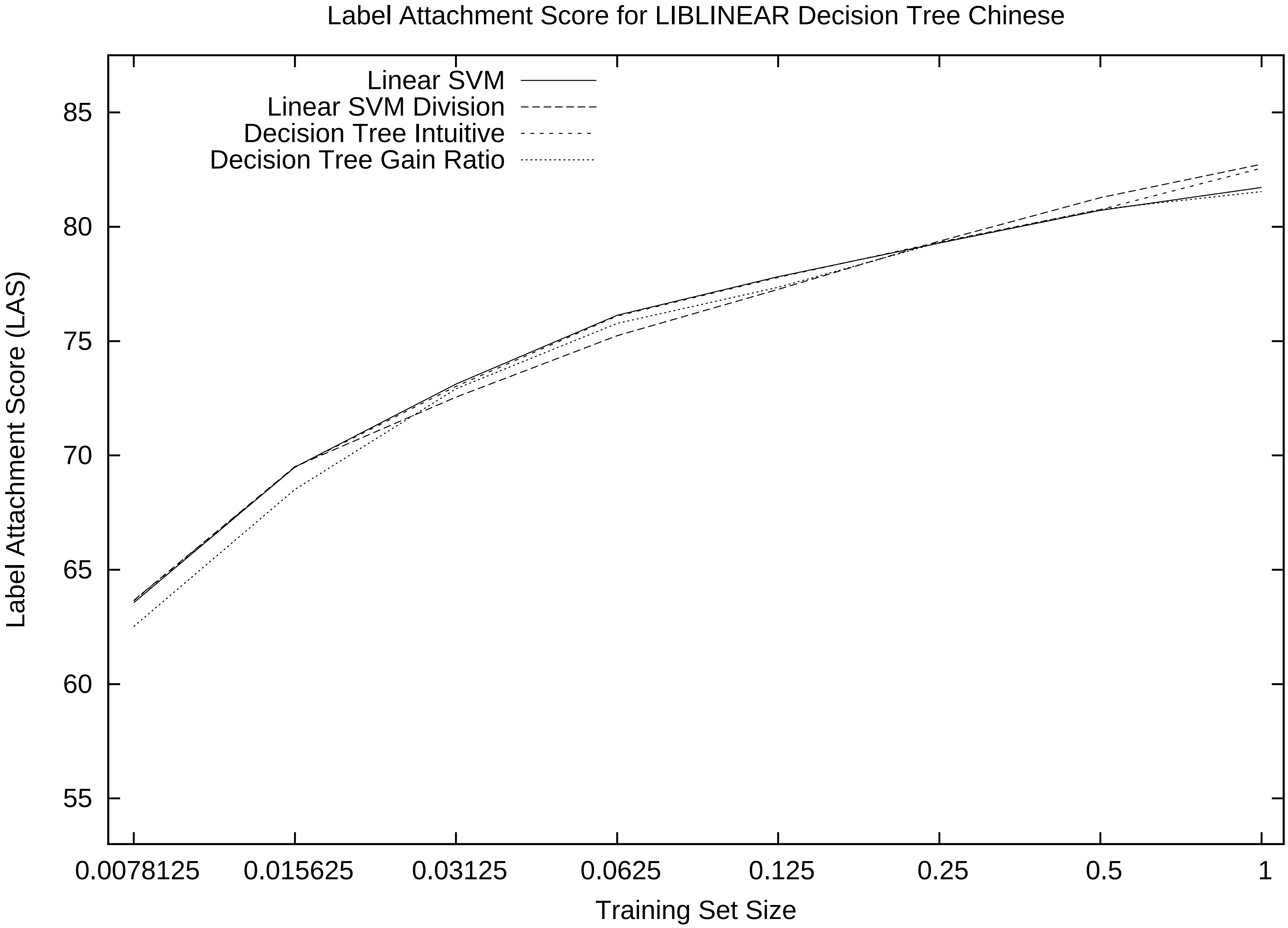

Figure A.8: The digram visualize the Label Attachment Score for all training set sizes created from the Chinese data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.1 and 4.8. Please, note that the x-axis in the diagram has logarithmic scale.

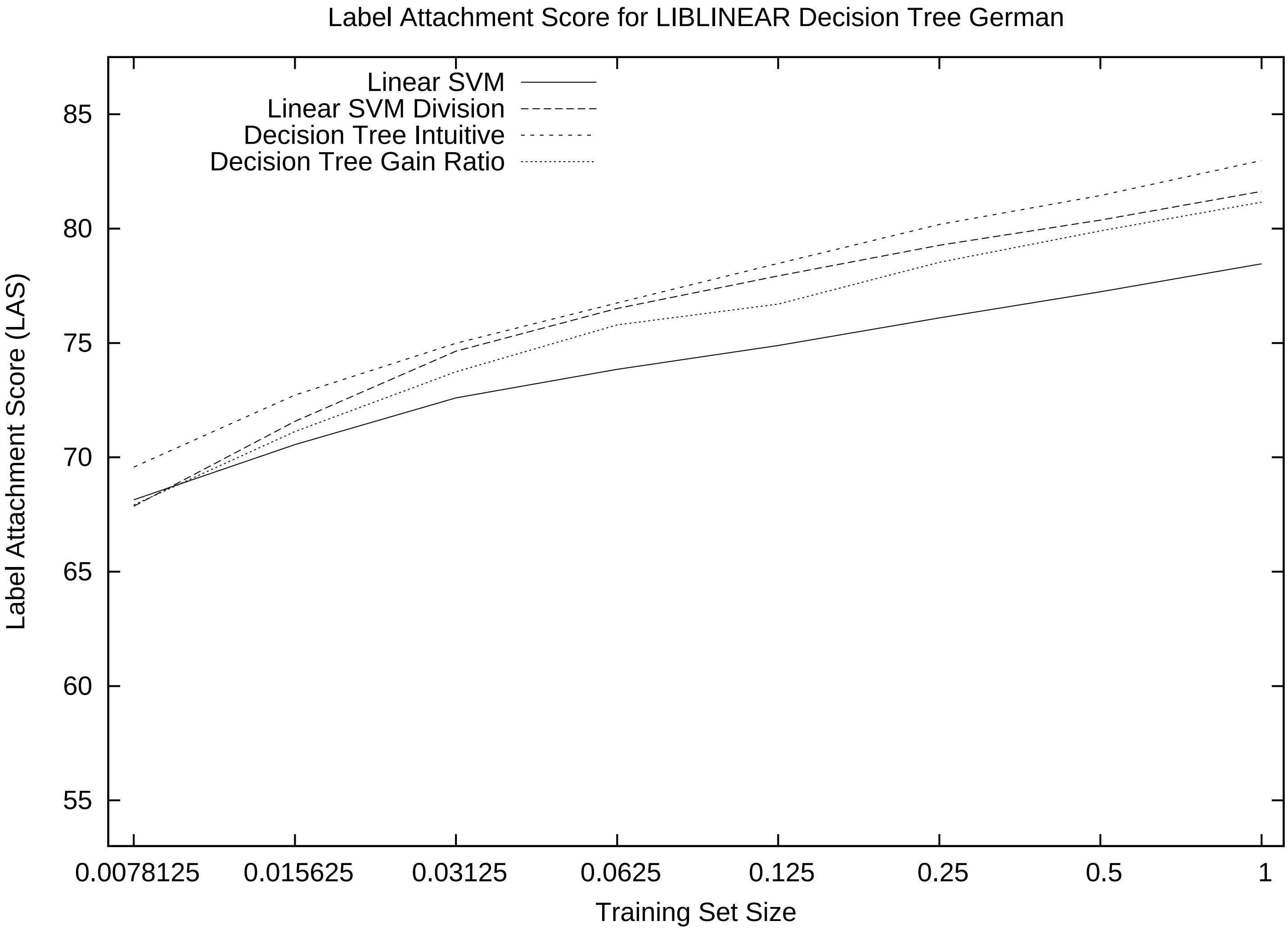

Figure A.9: The digram visualize the Label Attachment Score for all training set sizes created from the German data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.1 and 4.8. Please, note that the x-axis in the diagram has logarithmic scale.

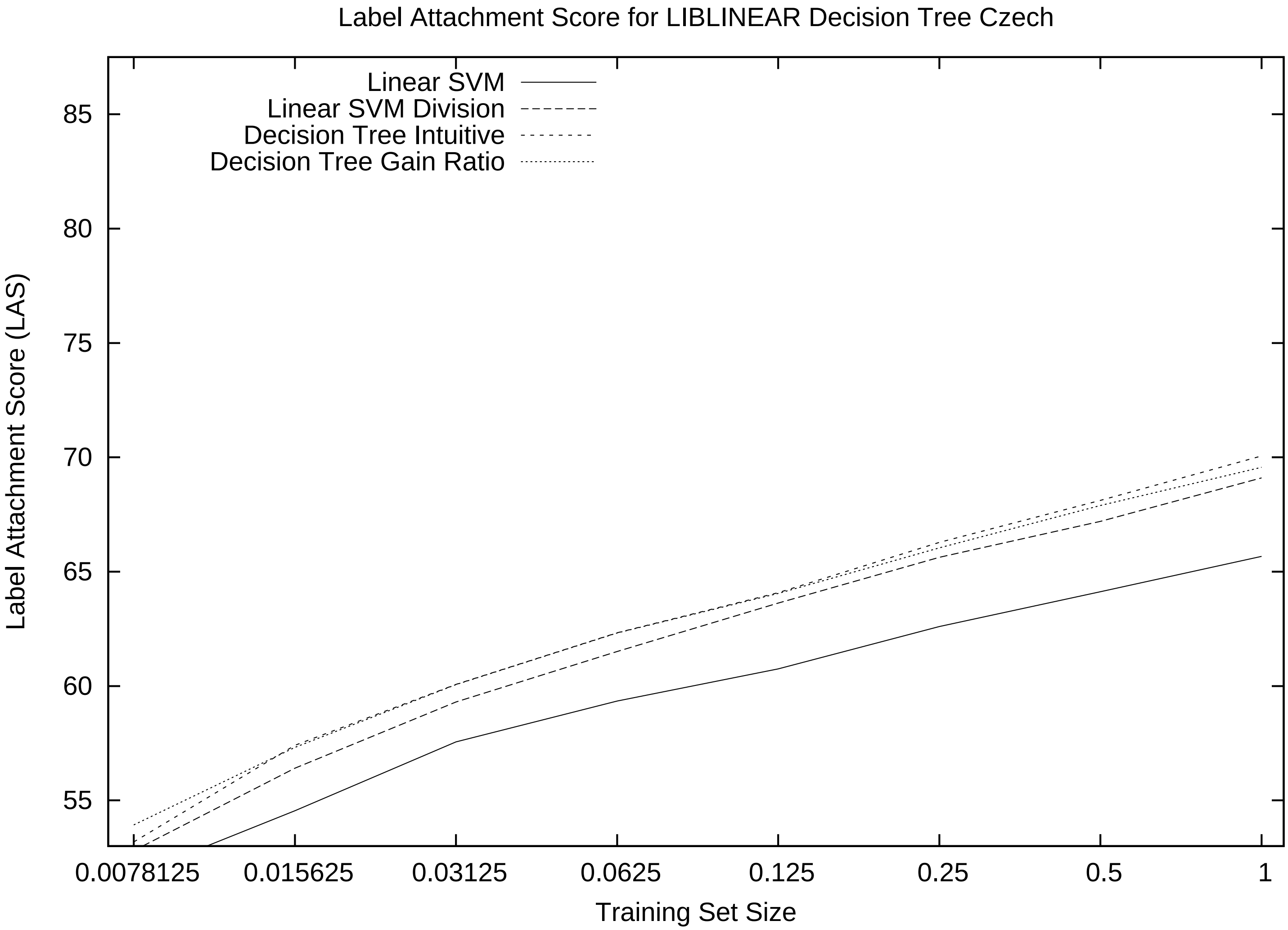

Figure A.10: The digram visualize the Label Attachment Score for all training set sizes created from the Czech data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.1 and 4.8. Please, note that the x-axis in the diagram has logarithmic scale.

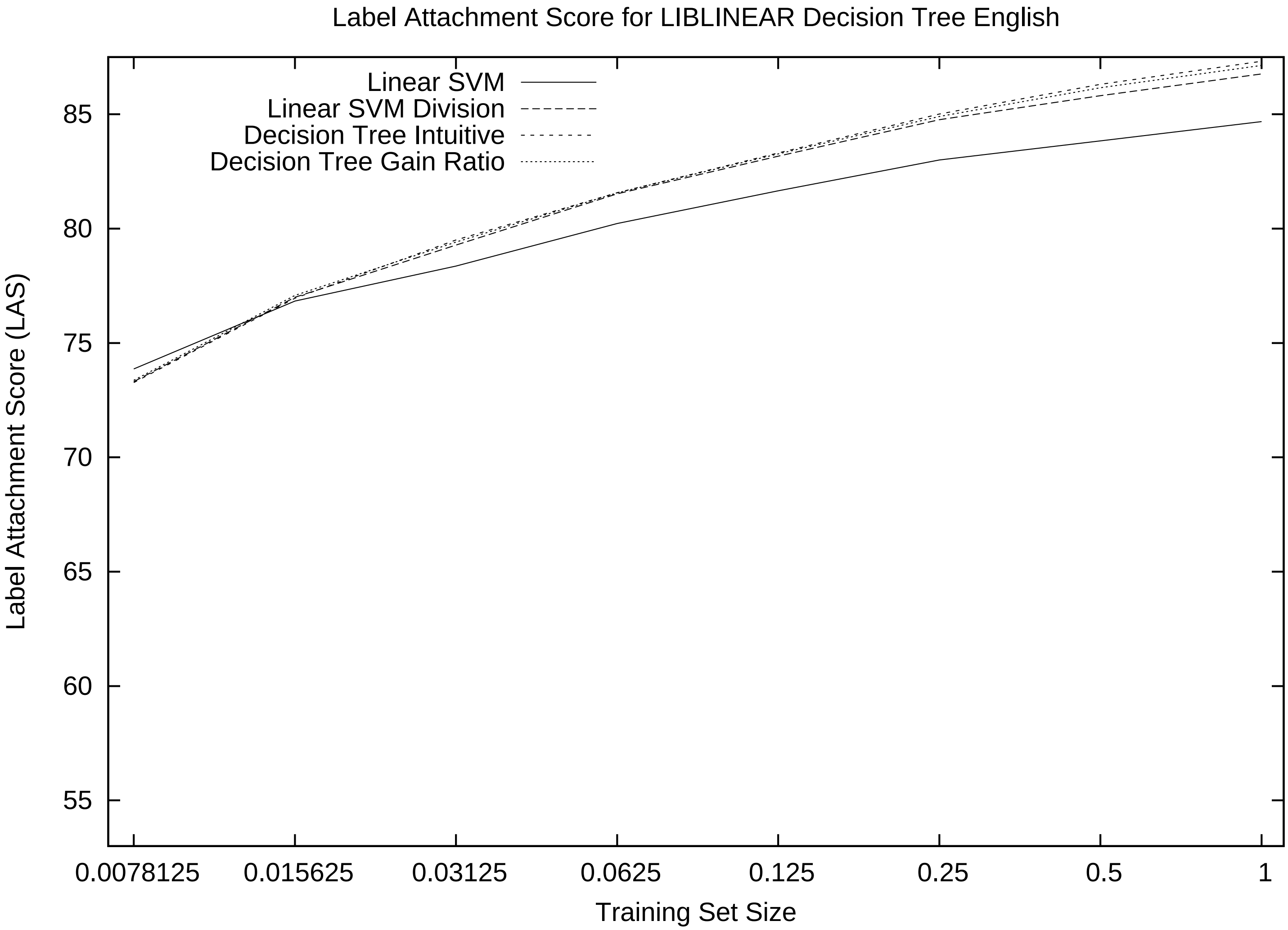

Figure A.11: The digram visualize the Label Attachment Score for all training set sizes created from the English data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.1 and 4.8. Please, note that the x-axis in the diagram has logarithmic scale.

Appendix B

MaltParser Settings

This chapter presents the settings used for MaltParser in the different experiments. The experiments presented in section 4.1 and 4.7 make use of MaltParser directly. The other experiments makes use of MaltParser's feature extraction system to get the training data in a readable format for LIBLINAR and LIBSVM. The full configuration files for the experiments presented in this report can be found at "http://github.com/kjellwinblad/master-thesis-matrial".

B.1 Basic Configuration

The following XML[1] is the content of the configuration file used in the experiment described in section 4.1 for the test that tests LIBLINAR together with division.

```
<?xml version="1.0" encoding="UTF-8"?>
<experiment>
 <optioncontainer>
  <optiongroup groupname="singlemalt">
   <option name="c" value="nivreeager"/>
  </optiongroup>
  <optiongroup groupname="nivre">
   <option name="root_handling" value="normal"/>
  </optiongroup>
  <optiongroup groupname="liblinear">
   <option name="liblinear_options" value="-s_4_-c_0.1"/>
  </optiongroup>
  <optiongroup groupname="guide">
   <option name="learner" value="liblinear"/>
   <option name="data_split_column" value="POSTAG"/>
   <option name="data_split_structure" value="Input[0]"/>
   <option name="data_split_threshold" value="1000"/>
   <option name="features" value="featuremodel.xml"/>
  </optiongroup>
 </optioncontainer>
```

[1] XML is a shorthand for Extended Markup Language and is commonly used as a way of structuring content of configuration files.

```
</experiment>
```

The option named `nivreeager` configures that Nivre-Arc-eager parsing algorithm is used. The option `root_handling` decides that normal root handling will be used in the Nivre-Arc-eager algorithm. The option named `liblinear_options` specifies which parameters that will be passed to LIBLINEAR. The options named `data_split_column`, `data_split_structure` and `data_split_threshold` configures that the training data will be split by the feature representing the property **POSTAG** of the word found in **Input[0]** (the first word is the buffer) and that all training sets created by division with number of training instances less than **1000** will be put in a special training set.

The option named `features` defines which file will be used as feature extraction model. The feature extraction configuration used in presented here:

```
<?xml version="1.0" encoding="UTF-8"?>
<featuremodels>
 <featuremodel name="nivreeager">
  <feature>InputColumn(POSTAG, Stack[0])</feature>
  <feature>InputColumn(POSTAG, Input[0])</feature>
  <feature>InputColumn(POSTAG, Input[1])</feature>
  <feature>InputColumn(POSTAG, Input[2])</feature>
  <feature>InputColumn(POSTAG, Input[3])</feature>
  <feature>InputColumn(POSTAG, Stack[1])</feature>
  <feature>OutputColumn(DEPREL, Stack[0])</feature>
  <feature>OutputColumn(DEPREL, ldep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL, rdep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL, ldep(Input[0]))</feature>
  <feature>InputColumn(FORM, Stack[0])</feature>
  <feature>InputColumn(FORM, Input[0])</feature>
  <feature>InputColumn(FORM, Input[1])</feature>
  <feature>InputColumn(FORM, head(Stack[0]))</feature>
 </featuremodel>
</featuremodels>
```

A detailed description of how the feature extraction model can be read is found in the documentation for MaltParser.

The configuration for the experiment described in section 4.1 when LIBLINEAR was run without any division of the training data is exactly the same as the configuration described in section B.1 with the difference that the `data_split` options are not used.

The configuration for the corresponding tests described above but with LIBSVM instead of LIBLINAR has the LIBLINEAR options replaced with corresponding LIBSVM options.

B.2 Advanced Feature Extraction Models for Czech and English

The feature extraction models presented in the following subsection are for *English* and *Czech* in combination with the two dependency parsing algorithms stack projection and stack lazy. The MaltParser option `parsing_algorithm` need to be set to `stackproj` for the stack projection feature extraction models and to `stacklazy` for the stack lazy feature extraction models.

B.2.1 English Stack Projection

```
<?xml version="1.0" encoding="UTF-8"?>
<featuremodels>
 <featuremodel>
  <feature>InputColumn(POSTAG, Stack[0])</feature>
  <feature>InputColumn(POSTAG, Stack[1])</feature>
  <feature>InputColumn(POSTAG, Stack[2])</feature>
  <feature>InputColumn(POSTAG, Stack[3])</feature>
  <feature>InputColumn(POSTAG, Lookahead[0])</feature>
  <feature>InputColumn(POSTAG, Lookahead[1])</feature>
  <feature>InputColumn(POSTAG, Lookahead[2])</feature>
  <feature>InputColumn(POSTAG, ldep(Stack[0]))</feature>
  <feature>InputColumn(POSTAG, ldep(Stack[1]))</feature>
  <feature>InputColumn(POSTAG, rdep(Stack[0]))</feature>
  <feature>InputColumn(POSTAG, rdep(Stack[1]))</feature>
  <feature>InputColumn(LEMMA, Stack[0])</feature>
  <feature>InputColumn(LEMMA, Stack[1])</feature>
  <feature>InputColumn(LEMMA, Stack[2])</feature>
  <feature>InputColumn(LEMMA, Lookahead[0])</feature>
  <feature>InputColumn(LEMMA, Lookahead[1])</feature>
  <feature>InputColumn(FORM, Stack[0])</feature>
  <feature>InputColumn(FORM, Stack[1])</feature>
  <feature>InputColumn(FORM, Lookahead[0])</feature>
  <feature>OutputColumn(DEPREL, ldep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL, ldep(Stack[1]))</feature>
  <feature>OutputColumn(DEPREL, rdep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL, rdep(Stack[1]))</feature>
 </featuremodel>
</featuremodels>
```

B.2.2 English Stack Lazy

```
<?xml version="1.0" encoding="UTF-8"?>
<featuremodels>
 <featuremodel>
  <feature>InputColumn(POSTAG, Stack[0])</feature>
  <feature>InputColumn(POSTAG, Stack[1])</feature>
  <feature>InputColumn(POSTAG, Stack[2])</feature>
  <feature>InputColumn(POSTAG, Stack[3])</feature>
  <feature>InputColumn(POSTAG, Lookahead[0])</feature>
  <feature>InputColumn(POSTAG, Lookahead[1])</feature>
  <feature>InputColumn(POSTAG, Lookahead[2])</feature>
  <feature>InputColumn(POSTAG, Input[0])</feature>
  <feature>InputColumn(POSTAG, ldep(Stack[0]))</feature>
  <feature>InputColumn(POSTAG, ldep(Stack[1]))</feature>
  <feature>InputColumn(POSTAG, rdep(Stack[0]))</feature>
  <feature>InputColumn(POSTAG, rdep(Stack[1]))</feature>
  <feature>InputColumn(LEMMA, Stack[0])</feature>
  <feature>InputColumn(LEMMA, Stack[1])</feature>
  <feature>InputColumn(LEMMA, Stack[2])</feature>
  <feature>InputColumn(LEMMA, Lookahead[0])</feature>
  <feature>InputColumn(LEMMA, Lookahead[1])</feature>
  <feature>InputColumn(FORM, Stack[0])</feature>
  <feature>InputColumn(FORM, Stack[1])</feature>
  <feature>InputColumn(FORM, Lookahead[0])</feature>
  <feature>OutputColumn(DEPREL, ldep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL, ldep(Stack[1]))</feature>
  <feature>OutputColumn(DEPREL, rdep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL, rdep(Stack[1]))</feature>
 </featuremodel>
</featuremodels>
```

B.2.3 Czech Stack Projection

```
<?xml version="1.0" encoding="UTF-8"?>
<featuremodels>
 <featuremodel>
  <feature>InputColumn(POSTAG,Stack[0])</feature>
  <feature>InputColumn(POSTAG,Stack[1])</feature>
  <feature>InputColumn(POSTAG,Stack[2])</feature>
  <feature>InputColumn(POSTAG,Stack[3])</feature>
  <feature>InputColumn(POSTAG,Lookahead[0])</feature>
  <feature>InputColumn(POSTAG,Lookahead[1])</feature>
  <feature>InputColumn(POSTAG,Lookahead[2])</feature>
  <feature>InputColumn(POSTAG,Lookahead[3])</feature>
  <feature>InputColumn(POSTAG,Lookahead[4])</feature>
  <feature>InputColumn(POSTAG,pred(Stack[0]))</feature>
  <feature>Split(InputColumn(FEATS,Stack[0]),\|)</feature>
  <feature>Split(InputColumn(FEATS,Stack[1]),\|)</feature>
  <feature>Split(InputColumn(FEATS,Stack[2]),\|)</feature>
  <feature>Split(InputColumn(FEATS,Lookahead[0]),\|)</feature>
  <feature>Split(InputColumn(FEATS,Lookahead[1]),\|)</feature>
  <feature>OutputColumn(DEPREL,ldep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL,rdep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL,ldep(Stack[1]))</feature>
  <feature>OutputColumn(DEPREL,rdep(Stack[1]))</feature>
  <feature>InputColumn(FORM,Stack[0])</feature>
  <feature>InputColumn(FORM,Stack[1])</feature>
  <feature>InputColumn(FORM,Lookahead[0])</feature>
  <feature>InputColumn(FORM,Lookahead[1])</feature>
  <feature>InputColumn(FORM,Lookahead[2])</feature>
  <feature>InputColumn(FORM,pred(Stack[0]))</feature>
  <feature>InputColumn(LEMMA,Stack[0])</feature>
  <feature>InputColumn(LEMMA,Stack[1])</feature>
  <feature>InputColumn(LEMMA,Lookahead[0])</feature>
  <feature>InputColumn(LEMMA,Lookahead[1])</feature>
  <feature>InputColumn(LEMMA,pred(Stack[0]))</feature>
 </featuremodel>
```

```
</featuremodels>
```

B.2.4 Czech Stack Lazy

```
<?xml version="1.0" encoding="UTF-8"?>
<featuremodels>
 <featuremodel>
  <feature>InputColumn(POSTAG,Stack[0])</feature>
  <feature>InputColumn(POSTAG,Stack[1])</feature>
  <feature>InputColumn(POSTAG,Stack[2])</feature>
  <feature>InputColumn(POSTAG,Stack[3])</feature>
  <feature>InputColumn(POSTAG,Input[0])</feature>
  <feature>InputColumn(POSTAG,Lookahead[0])</feature>
  <feature>InputColumn(POSTAG,Lookahead[1])</feature>
  <feature>InputColumn(POSTAG,Lookahead[2])</feature>
  <feature>InputColumn(POSTAG,Lookahead[3])</feature>
  <feature>InputColumn(POSTAG,Lookahead[4])</feature>
  <feature>InputColumn(POSTAG,pred(Stack[0]))</feature>
  <feature>Split(InputColumn(FEATS,Stack[0]),\|)</feature>
  <feature>Split(InputColumn(FEATS,Stack[1]),\|)</feature>
  <feature>Split(InputColumn(FEATS,Stack[2]),\|)</feature>
  <feature>Split(InputColumn(FEATS,Lookahead[0]),\|)</feature>
  <feature>Split(InputColumn(FEATS,Lookahead[1]),\|)</feature>
  <feature>OutputColumn(DEPREL,ldep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL,rdep(Stack[0]))</feature>
  <feature>OutputColumn(DEPREL,ldep(Stack[1]))</feature>
  <feature>OutputColumn(DEPREL,rdep(Stack[1]))</feature>
  <feature>InputColumn(FORM,Stack[0])</feature>
  <feature>InputColumn(FORM,Stack[1])</feature>
  <feature>InputColumn(FORM,Lookahead[0])</feature>
  <feature>InputColumn(FORM,Lookahead[1])</feature>
  <feature>InputColumn(FORM,Lookahead[2])</feature>
  <feature>InputColumn(FORM,pred(Stack[0]))</feature>
  <feature>InputColumn(LEMMA,Stack[0])</feature>
  <feature>InputColumn(LEMMA,Stack[1])</feature>
  <feature>InputColumn(LEMMA,Lookahead[0])</feature>
  <feature>InputColumn(LEMMA,Lookahead[1])</feature>
  <feature>InputColumn(LEMMA,pred(Stack[0]))</feature>
 </featuremodel>
</featuremodels>
```

B.3 Configuration for Division and Decision Tree in Malt Parser

In the experiment described in section 4.7 two variants of the new MaltParser decision tree functionality are tested. One variant that uses the Gain Ratio measurement to calculate the division order and one that uses a predefined division order. Apart from the options that have a name starting with `data_split` the variants are exactly the same as the configuration example provided in appendix B.1. The variant that uses automatic division order can be derived by replacing the options containing data_split with the following options in that configuration:

```
<option name="tree_automatic_split_order" value="true"/>
<option name="tree_split_threshold" value="1000"/>
```

The other variant that uses a predefined division order can be obtained by instead replacing the options containing data_split with the following options:

```
<option name="tree_split_columns"
    value="POSTAG@POSTAG@POSTAG@POSTAG@POSTAG@POSTAG"/>
<option name="tree_split_structures"
    value="Input[0]@Stack[0]@Input[1]@Input[2]@Input[3]@Stack[1]"/>
<option name="tree_split_threshold"  value="50"/>
```