The Impact of Training Data Division in Inductive Dependency Parsing

Kjell Winblad

May 30, 2011
Master's Thesis in Computing Science, 30 credits
Supervisor at Department of Information Technology Uppsala: Olle Gällmo
External Supervisor at The Department of Linguistics and Philology
Uppsala: Joakim Nivre
Examiner: Anders Jansson

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

Abstract

Syntax parsing of natural language can be done with inductive dependency parsing which is also often referred to as data-driven dependency parsing. Data-driven dependency parsing makes use of a machine learning method to train a classifier that guides the parser. In this report an attempt to improve the state of the art classifier for data-driven dependency parsing is presented.

In this thesis work it has been found experimentally that division of the training data by a feature can increase the accuracy of a dependency parsing system when the partitions created by the division are trained with linear Support Vector Machines. It has also been shown that the training and testing time can be significantly improved and that the accuracy does not suffer much when this kind of division strategy is used together with nonlinear Support Vector Machines. The results of experiments with decision trees that use linear Support Vector Machines in the leaf nodes indicates that a small improvement of the accuracy can be gained with that technique compared to simply dividing by one feature and training the resulting partitions with linear Support Vector Machines.

Effekten av att dela träningsdata i induktiv dependensparsning

Sammanfattning

Syntaxanalys av naturligt språk kan göras med induktiv dependensparsning som ofta också kallas datadriven dependensparsning. Datadriven dependensparsning använder en maskininlärningsmetod för att träna en klassificerare som vägleder parsern. I den här rapporten presenteras ett försök att förbättra en av de bästa metoderna som finns idag för dependensparsning.

I det här examensarbetet har det visats i experiment att delning av träningsdata baserat på värdet av en egenskap hos träningsexemplen kan förbättra parsningsresultatet för ett dependensparsningssystem när delarna som har skapats vid uppdelningen tränas med linjära stödvektormaskiner. Det har också visats att tiden det tar att träna och parsa kan förbättras avsevärt och att parsningsresultatet inte förändras mycket när den här typen av uppdelningsstrategier används tillsammans med ickelinjära stödvektormaskiner. Resultaten från experiment med beslutsträd som använder linjära stödvektormaskiner i lövnoderna av beslutsträdet tyder på att en liten förbättring av parsningsresultatet kan fås med tekniken jämfört med att göra en enkel uppdelning av träningsdata och träna de resulterande delarna med linjära stödvektormaskiner.

Contents

Chapter 1

Introduction

To automatically generate syntax trees for sentences in a natural language text has several applications. It can be used in, for example, automatic translation systems and semantic analysis. A technique for generating such trees that has gained increased popularity in recent years is data-driven dependency parsing [KMN09]. The data-driven dependency parsing technique called transition-based dependency parsing works by building up a syntax tree sequentially by applying different transitions on the current parsing state. Which transition that shall be applied in a given state is decided by a function which is often called the oracle function. An oracle function is a function that given a parsing state outputs the next step to be taken in the parsing process. It is a very difficult problem to find a good oracle function due to the ambiguity of natural languages. One of the best methods known so far for creating the oracle function is a supervised machine learning technique called nonlinear Support Vector Machines (SVMs) [YM03]. Supervised machine learning techniques are algorithms that given training examples of input values and output values creates a model that can be used to predict output values from input values that may not exist in the training examples. If a syntax tree for a sentence makes sense or not can only be decided by humans. Therefore it is natural to deduce the training data for the machine learning technique used to create the oracle function from syntax trees created by humans.

The training phase of a nonlinear SVM is very memory and computationally expensive. To speed it up it is possible to divide the input data in a reproducible way and then train many smaller SVMs that can be combined to produce the final classifier [GE08]. 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 at Uppsala University found that the result was better with division than without. Investigating this is interesting, since the training and testing time of the state of the art transition-based parsing system could be reduced significantly, if the division technique could be refined to give similar accuracy as the nonlinear SVM. Furthermore, it could give insight into the classification problem which could lead to other improvements.

This report contains experimentation and analysis with the aim to explain and confirm the improved result gained by using the division strategy together with linear SVMs described above. It also contains experiments with a more advanced tree division strategy. An implementation of the more advanced tree division strategy has been done as a plug in to the dependency parsing system MaltParser[1] [NM07].

The rest of this chapter describes the problems that are dealt with in this thesis work

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

and the goals of the thesis. Chapter 2 explains the technologies that are needed to be able to understand the results of this thesis, namely Dependency Parsing, SVMs and decision trees. Chapter 3 explains the hypotheses that are tested by the experiments and describes the software that has been used as well as the data sets. Chapter 4 goes through the experiments performed and discusses the aim of them as well as the results. Chapter 5 describes the implementation and usage of the new plugin created for MaltParser. Finally, chapter 6 discusses the achievements of the work as well as its limitations and possible future work.

1.1 Problem Description

This section 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 in chapter 6.

1.1.1 Problem Statement

The aim of the thesis work is to study dependency parsing and in particular the oracle function used to determine the next step in the parsing procedure. If it turns out that the experiments and studies of machine learning methods show that it could be useful to implement a new feature 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:

Goal number 1 is to find out more in detail than what previously has been done how division of training data 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 performs inside the dependency parsing system in terms of measures such as percentage of correct labels compared to a correct syntax tree). This is interesting because it is known that division in some cases has a positive effect on the accuracy, but it is not very clear in which situations it has a positive effect on the accuracy. More insight into this can lead to new implementations of the oracle function which may have faster training time as well as acceptable or possibly even better accuracy than what has been obtained so far.

Goal number 2 is to do an analysis about the theoretical reason for the effect that the division has. This goal can be seen as a subgoal of *goal number 1*. The difference between this goal and *goal number 1* is that this goal puts more emphasis on why division effects the accuracy. Whereas *goal number 1* is more about in what way division effects the training. For example, how much different is the accuracy when division is used from when no division is used. This may lead to new ideas about how to improve the accuracy of the classifier as well as new insights into the characteristics of the classification problem.

Goal number 3 is to implement an alternative oracle function into MaltParser, if the investigations described above shows that it could be useful. The difference in training

time for a linear SVM and a nonlinear SVM is large. A shorter training time could be of help when parsing methods are tested. As an example, a training time for a nonlinear SVM on a dependency parsing problem can be about a week when the linear version of the classifier can be trained within a few hours. Therefore, it could be useful with a new type of oracle function if it has a faster training time even if its accuracy is not better than the state of the art.

Chapter 2

Background

A dependency parser is a system that parses sentences from a natural language into tree structures that belong to a dependency grammar. Dependency parsers often make use of machine learning methods to guide the parser. One of the machine learning methods that have shown the best results is Support Vector Machines (SVMs). How all these concepts fit together will be explained in the rest of the sections in this chapter.

Dependency Parsing is studied in the research fields computational linguistics and linguistics. SVMs are studied in the research field machine learning.

2.1 Dependency Grammar

A dependency grammar like other grammatical frameworks describe the syntactic structure of sentences. Dependency grammar differs from other grammatical frameworks because the syntax is described as directed graphs, where the labeled edges represents dependencies between words. The graph in figure 2.1 gives an example of such a structure.

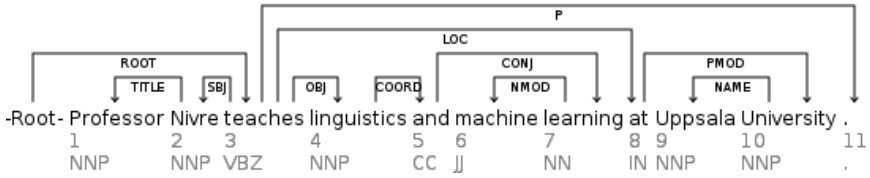

Figure 2.1: The figure shows the dependency grammar structure for a sentence. For example, the node University has an edge labeled NAME to the word Uppsala, which describes a dependency relation between University and Uppsala. [1]

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 with other relations to form a connected tree. It is possible to build hundreds of different dependency trees for a normal sentences, but just a few of them will make sense semantically to a human. Due to the complexity of natural languages and the ambiguity

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

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 systems. 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 systems have both a grammar based component and a machine learning component. For example a grammar based system can make use of a data-driven approach to generate the formalized grammar and some grammar based systems generate many candidate dependency graphs from a grammar and use a machine learning technique to select one of them. [KMN09]

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. The parsing algorithm described here is a bit simpler to give an an understanding of the method without going into unnecessary details. The parsing algorithm used in the experiments is called Nivre Arc-eager [Niv08].

A transition-based parsing system contains the three components:

A configuration that contains the current state of the parsing system.

A set of rules, where every rule transforms a configuration to another configuration.

An oracle function that given a configuration outputs a rule to apply.

The components differ in different variants of transition-based systems, but the basic principle is the same. 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 configuration that consists of the three components:

- A set of labeled arcs T from one word to another word. The set is empty in the initial state.
- A stack S containing words that are partially parsed. The stack only contains an artificial word called ROOT in the initial configuration. The artificial word ROOT is of course not the same as the ordinary word "root". The artificial word ROOT is added for convenience. The ROOT word will always be the root of the parsed tree.
- A buffer B 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 words in the sentence to be parsed with the first word of the sentence at the first position in the buffer and the second word in the sentence at the second position in the buffer etc. The configuration in which the buffer is empty defines the end of the parsing.

The following instructions are used in basic system to change the configuration:

Pop w **from** S means that the first word in the stack w shall be removed from the stack S.

Add the arc (w_1, l, w_2) **to** T means that an arc from the word w_1 to the word w_2 with the label l shall be added to the set of arcs T.

Replace the first word w_1 **in** B **with** w_2 means that the first word w_1 in the buffer B shall be replaced with the word w_2.

Remove the first word w **in** B means that the first word w in the buffer B shall be removed from the buffer B.

Push w **to** S means that the word w shall be pushed to the top of the stack S.

The following list describes the set of rules used in the basic system with names of the rules and the instructions that shall be performed on the configuration when the rules are used:

LEFT-ARC(l): Pop w_1 **from** S and **add the arc** (w_2, l, w_1) **to** T, where w_2 is the first word in B. A precondition for this rule to be allowed to be applied is that w_1 is not the special word ROOT. This is to prevent the word ROOT from depending on any other words.

RIGHT-ARC(l): Pop w_1 **from** S, **replace the first word** w_2 **in** B **with** w_1 and **add the arc** (w_1, l, w_2) **to** T.

SHIFT: Remove the first word w **in** B and **push** w **to** S.

The parsing algorithm works by applying the rules until the buffer is empty. Then if the arcs in the arc set are not connected or if words are missing from the sentence a tree containing all words is constructed by attaching words to the special ROOT word. Both the parsing algorithm described here and the Nivre Arc-eager system used in the experiments have been proven to be both sound and complete, which means that parsing will always result in a forest of dependency trees from which a single dependency tree can easily be created by attaching the trees to the special ROOT word and all possible projective trees[1] 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 a 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. The selection of features are often done by people with a lot of domain specific knowledge. One of the most successful machine learning methods that have been used for oracle approximation is Support Vector

[1] See section 2.1.2 in [KMN09] for an explanation of what it means for a dependency tree to be projective.

Machines (SVMs) with the Kernel Trick also called nonlinear SVMs. The general idea for how SVMs work is explained in section 2.4.

Given that the oracle function approximation runs in constant time, it has been proven that both the basic system and Nivre arc-eager can parse sentences with the time complexity $O(N)$, where N is the length of the sentence [Niv08].

2.3 Measuring the Accuracy of Dependency Parsing Systems

The Labeled Attachment Score (LAS) is a commonly used measurement used to evaluate dependency parsing systems that is used in the experiments presented in chapter 4. The LAS is the percentage 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 it depends on.

Other measures that are commonly used is Unlabeled Attachment Score which is the same as LAS but without any check of the label, and the Exact Match Measure that is the percentage of sentences that exactly match the reference sentences.

2.4 Support Vector Machines

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

Because the exact description of how SVMs work is such a complex topic involving advanced mathematical concepts, only the most fundamental idea behind it and the concepts necessary to understand the results of the the experiments described in this report 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.

2.4.1 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-dimensional space that separate the square class from the circle class with the maximum margin. The two parallel lines illustrate the borders of the margin which should be maximized.

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 sets are not linearly separable but the most basic SVM can be extended to support that by making a trade off between the distance from the separation hyperplane to the margin and the amount of misclassified training instances. [TSK05]

2.4.2 The Kernel Trick

The kernel trick is the name of a method to transform the input space to a space with higher dimensionality. This is an effective technique to improve the accuracy of classifiers when

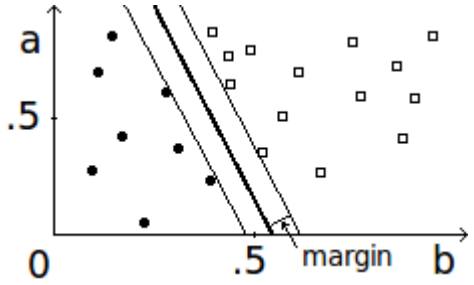

Figure 2.2: The maximum margin hyperplane that separates two classes in 2-dimensional space.

the classification problem is not linearly separable. It has been shown that the accuracy of dependency parsing can be significantly improved if the linear SVM used as oracle function is replaced by an SVM that makes use of the Kernel Trick. Due to the higher dimensionality when the Kernel Trick is used both the training and testing time is much longer. For an experimental comparison between the time complexity of systems that use the Kernel Trick and systems that do not, 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.

2.4.3 The Extension to Support Multiple Class Classification

The basic SVM only supports classification to one of two classes. There are many extensions that allow 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 the rest extension. It works by first creating one internal SVM for every class and then trains each internal SVM using one class for the class it represents and the 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 for example 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].

2.5 Decision Trees

Decision trees is an alternative to SVMs for classification that also can be combined with SVMs or other machine learning methods to get improved results [SL91]. The basic idea behind decision trees is to divide a hard decision until there is only one class or a high probability for one class left. In the training phase of a decision tree a tree structure is built where the leaf nodes represents final decisions and the other nodes represents divisions of the original classification problem. As an example consider the dependency parsing system described in section 2.2. Also consider a feature extraction model that extracts the type of the word on the top of the stack and the type of the first word in the buffer.

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

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: Training examples for the decision tree example.

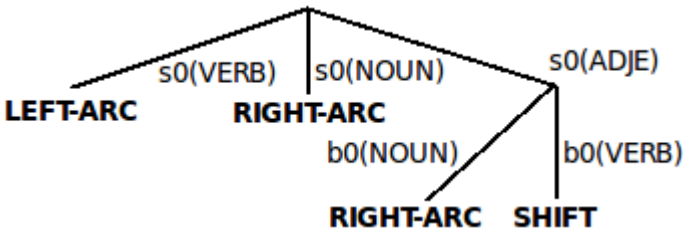

Figure 2.3: The figure shows an example of a decision tree, where $s0(X)$ represent that the word on top of the stack has the word type X and $b0(X)$ that the first word in the buffer has X as word type.

classified by the decision tree by first going down from the root node in the tree along the branch marked s0(ADJE) and then following the branch marked v0(VERB) where it would be classified to be of the **SHIFT** class because there is a child node marked **SHIFT** 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. Another approach is to use another machine learning technique as for example an SVM to train a subclassifier 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, 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.

2.5.1 Gain Ratio

One problem when creating decision trees is how to know which feature is best to divide on in a particular branch. It is desirable to have few nodes in the tree to reduce generalization error and at the same time to have enough nodes to make use of all useful information. 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 knowledge gained from looking at one feature. A commonly used measurement is called information gain. Information gain looks at the impurity of 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 that are shallow and wide. 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 off 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 here 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:

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

, 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.

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:

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

, where t is the parent training set, d 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.

The Gain Ratio measurement reduces the value of information gain by dividing it with something that can be called Split Info as shown in equation 2.3:

$$gain\_ratio(t,s) = \frac{information\_gain(t,s)}{split\_info(t,s)} \tag{2.3}$$

Split Info gets a higher value when there are more distinct values of a particular feature. Equation 2.4 shows how the Split Info is calculated:

$$split\_info(t,s) = -\sum_{i=1}^{v} \frac{N(t_i)}{N(t)} log_2(\frac{N(t_i)}{N(t)}) \tag{2.4}$$

, 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.

For more detailed information about splitting strategies and alternative impurity measurements, see [TSK05] section 4.3.4.

2.6 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 into test sets with an equal number of instances in each. If the training set is divided into N test sets, the cross-validation is called N-fold cross-validation. For every test set created a training set is created by concatenating all other test sets. The cross-validation accuracy is calculated for every test set and corresponding training set, first train a classifier with the training set and then test the resulting classifier with the test set. The mean of all the tests is said to be the cross-validation accuracy.

Chapter 3

Hypotheses and Methodology

This chapter describes the hypotheses 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 that are tested in the experiments described in chapter 4.

1. When a linear Support Vector Machine (SVM) 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 previous experiments have indicated that dividing the training data can result in good accuracy [GE08].

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 linear SVM is not powerful enough to separate the classes in an optimal way, but a technique where an initial division is used to create several subproblems that can be solved by SVMs is more powerful in that sense.

3. The smaller the partitions of the division becomes the more accurate the individual subclassifiers 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.

3.2 Methods

The experiments described in section 4.1 and 4.7 made use of MaltParser. The rest of the experiments test different variants of machine learning methods with training and test data from the feature extraction step in MaltParser's training mode. This was done to eliminate as many irrelevant factors as possible and make the experiments easier to perform.

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 were 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[3].

3.3 Tools

Many different software tools have been used during the thesis work. The most important tools are presented in the following sections.

3.3.1 MaltParser

MaltParser is an open source data-driven transition-based dependency parsing system [NM07]. It is written in the Java programming language. It has been proven to be one of the best performing systems by getting one of the top scores in the competition CoNLL Shared Task 2007 on Dependency Parsing [NHK+07]. The system is very configurable which makes it possible to optimize for different languages. The system is 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 of that mode 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 eval07.pl[4] has been used.

3.3.2 The Support Vector Machine Libraries LIBSVM and LIBLINEAR

LIBLINEAR and LIBSVM[5] are two SVM implementations that are integrated into MaltParser. LIBLINEAR implements linear SVMs and LIBSVM implements nonlinear SVMs [CL01, FCH+08]. 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.

The settings for the two libraries used in the experiments are presented in appendix B.3.

[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.

[3] All scripts and configurations can be found at the following location "http://github.com/kjellwinblad/master-thesis-matrial".

[4] The measurement tool eval07.pl can be found in the dependency parsing wiki "http://depparse.uvt.nl/depparse-wiki/SoftwarePage".

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

3.4 Data Sets

The training data sets used in the experiments described in chapter 4 are in the CoNLL Shared Task [NHK+07] 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 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	Words	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+]
English	39279	848743	CoNLL-2009 shared task [HCJ+09]

Table 3.1: Treebanks used in the experiments described in chapter 4.

Chapter 4

Experiments

In the following sections, the experiments conducted in this work are presented. The experiments are related to the hypotheses presented in section 3.1. The *Results and Discussion* sections in this chapter often refer to the different hypotheses. The experiments can be seen as dependent on each other because after an experiment was finished, the next experiment to be conducted was decided based on the results of the previous experiments. The experiments were conducted in the same order as they are presented here.

4.1 Division of the Training Set With Linear and Nonlinear SVMs

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 a linear SVM (LIBLINEAR) or a nonlinear SVM (LIBSVM) as learning method. The experiment was done with three different languages to see if the results are the language dependent.

The following training methods were tested in the experiment:

- Linear SVM with division of the training set
- Linear SVM without division of the training set
- Nonlinear SVM with division of the training set
- Nonlinear SVM without division of the training set

When division was used the training data was divided by the feature representing the POSTAG[1] property of the first element in the buffer. A test set was picked out from the original data set containing 10% of the instances. Eight different training sets 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 same training and testing sets were used for all four training methods. The MaltParser configuration used in the experiment is explained in appendix B.1.

[1] POSTAG is the name of a column in the CoNLL data format used to represent sentences. In the POSTAG column a value representing fine-grained part-of-speech for the word can be found. The set of values that can be used for that column is language dependent.

4.1.1 Results and Discussion

		Linear SVM							
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: The table contains 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** = *Labeled 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 SVMs compared to the ones using linear SVMs. It can also be seen that division gives better training time than without division when nonlinear SVMs are used. 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 faster than linearly 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. The theoretical time complexity for the case with division is not worse than the case without division so this must be because of an "external" factor such as the increase in reading from disk caused by more models.

Diagrams displaying the Labeled Attachment Score (LAS) for the tests with the three languages can be seen in figure A.1, A.2 and A.3 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 largest training sets, the accuracy is very similar with and without division when nonlinear SVM is used, but there is clear difference in accuracy with and without division for the linear SVM. 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 larger the training set is.

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

		Nonlinear SVM							
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: The table contains 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 largest 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** = *Labeled Attachment Score*

accuracy. A possible explanation of why the same difference does not exist for nonlinear SVMs can be that it is more powerful than the linear SVMs and hence can handle the harder undivided problem better than the linear classifier and therefore, the nonlinear SVMs can not get the same improvement from division. This would 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 its 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 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 Labeled Attachment Score (LAS) used when measuring the accuracy of parsing. The LAS measurement measure a parsing system that makes use of a machine learning method which can be measured with cross validation accuracy. The measures can not automatically be translated to each other because a wrong classification by the machine learning method may result in several errors in the sentence that is parsed. 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 its cross validation accuracy. This is illustrated in the figure A.4, A.5 and A.6 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 largest partitions and among small ones. The median based box plots presented in figure 4.1 show how the accuracy vary among the partitions.

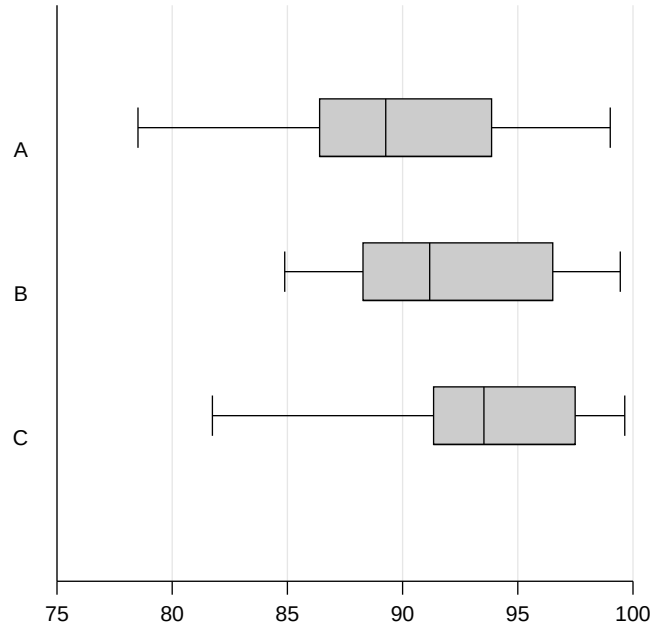

Figure 4.1: The diagram shows three median based box plots for the accuracy of the partitions created by division. Plot A is for Swedish, B is for Chinese and C is for German.

A possible explanation of why 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. For example most instances of those partitions may belong to just a few classes that are easy to separate. This would supports *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. This may imply 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 Another Division of the Worst Performing Partitions

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 large, which gives a reason to believe that the division was not the best for all partitions and that another 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 SVM (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.

	Language	**Size**	**Feat. 1**	**Feat. 2**	**No div.**
Worse Than Average	**Swedish**	0.52	86.90	86.25	86.82
	Chinese	0.64	89.50	89.39	89.57
	German	0.39	88.10	87.91	85.84
Better Than Average	**Swedish**	0.48	95.72	95.59	95.72
	Chinese	0.36	96.75	96.43	96.61
	German	0.61	95.54	95.41	94.93
Everything	**Swedish**	1.0	91.15	90.75	90.92
	Chinese	1.0	92.09	91.72	92.04
	German	1.0	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.

4.3.1 Results and Discussion

The results from the execution of the experiment is presented in table 4.3. Division on feature 1 generally gives 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 where the accuracy is a little bit worse with division than without. Perhaps the worst performing partitions are hard to separate independently of which division feature is chosen. It is also possible that feature 2 is very similar to feature 1. Another division feature could give another result so more division features need to be tested to make sure.

4.4 Different Levels of Division

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

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 greater 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 0.73%. *Swedish* is the smallest training set which could explain 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 were greater for the first division than the second supports *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 used for division in the first and second division are good. It is possible that other features are better for the divisions and that might

	No Div.	Sign.	1 Div.	Sign.	2 Div.
Swedish	90.916	$<$ (70%)	91.150*	$>$ (55%)	90.979
Chinese	92.044	$<$ (21%)	92.089	$<$ (36%)	92.168*
German	91.039	$<$ (99%)	92.678	$<$ (22%)	92.708*
Czech Stack Lazy	89.979	$<$ (99%)	91.175	$<$ (99%)	91.852*
Czech Stack Projection	89.783	$<$ (99%)	91.016	$<$ (99%)	91.616*
English Stack Lazy	94.374	$<$ (99%)	94.756	$<$ (99%)	94.954*
English Stack Projection	94.400	$<$ (99%)	94.763	$<$ (99%)	95.008*
Average	*91.791*		*92.518*		*92.755**

Table 4.4: The table shows weighted cross validation scores for the different levels of division. The columns with the header "Sign." shows the statistical significance of the difference between the divisions. For example the statistical certainty that one division gives better accuracy than no divisions for a Swedish data set of the size used in the experiment is greater than 70%. In other words, an element in a "Sign." column shows the statistical confidence that there is a difference in accuracy between the method used to get the value to the left of the element and the method used to get the value to the right of the element. The estimation of the statistical certainty is based on the assumption that the cross validation accuracy has equal or better certainty than a test with a single test set of the same size as the test sets used in the cross validation[1].

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. The aim of this experiment is 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 made by the supervisor of the thesis project Joakim Nivre during his research. The list is presented in table 4.5.

The decision tree was created with the algorithm presented in listing 1. 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 size.

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.

[1] How the confidence intervals are calculated can be seen at the following location "https://github.com/kjellwinblad/master-thesis-matrial/blob/master/scala_code/tools/ConfidenceIntervallCalculation.scala".

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

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.

4.5.1 Results and Discussion

	Intuitive	Sign.	2 Div.	Sign.	Gain Ratio
Swedish	91.168	$>$ (60%)	90.979*	$>$ (11%)	90.947
Chinese	92.118	$<$ (23%)	92.168*	$>$ (15%)	92.135
German	93.132*	$>$ (99%)	92.708	$<$ (96%)	92.936
Czech Stack Lazy	91.866	$>$ (10%)	91.852	$<$ (63%)	91.947*
Czech Stack Projection	91.654	$>$ (27%)	91.616	$<$ (85%)	91.771*
English Stack Lazy	95.020	$>$ (65%)	94.954	$<$ (98%)	95.120*
English Stack Projection	95.077	$>$ (67%)	95.008	$<$ (96%)	95.158*
Average	*92.862**		*92.755*		*92.859*

Table 4.6: The accuracy for different languages 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. See section 4.6 for an explanation of the Gain Ratio column and the description of table 4.4 for a description of the "Sign." columns.

The results of the experiment are summarized in table 4.6. Compared to the average accuracy obtained from two divisions in the experiment described in section 4.4 the decision tree gives an improvement of about 0.1%. 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. 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. This could explain why *Chinese* got slightly worse result with the decision tree anyway.

Looking at the structure of the decision trees created for the different training sets, some nodes are divided more than others and the maximum depth for the trees seems to increase with the size of the training set[2].

[2]Images that show the structure of the created decision trees can be found at the location "http://github.com/kjellwinblad/master-thesis-

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 its 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

Listing 1: The decision tree algorithm used in the experiments.

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

4.6 Decision Tree With Division Order Decided by Gain Ratio

The experiment described in section 4.5 indicated that combining a decision tree 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 2.5.1. This experiment was set up to try the Gain Ratio as ordering measurement for the possible division features.

matrial/tree/master/configs_and_scripts/exp3MainDir/results/graphs".

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 4.6. The average accuracy for the data sets trained with the Gain Ratio and intuitive decision order are almost the same. The difference is only 0.003%.

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 just 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 experiment described in section 4.1. Also the dependency parsing algorithm and feature extraction model are the same as in section 4.1. The minimum partition size was set to 50. 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 larger than 50. For comparison the linear SVM with division tests 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 and table 4.7. Diagrams displaying the Labeled 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 confirm the results of the previous experiments. The accuracy can get better for most languages with a decision tree than division on just one feature. The intuitive division order has better accuracy for all data sets compared to the Gain Ratio generated division order.

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 created consume about the same

	Intuitive	Sign.	Division	Sign.	Gain Ratio
Swedish	82.18	$<$ (19%)	82.28	$>$ (40%)	82.06
Chinese	82.56	$>$ (5%)	82.54	$>$ (99%)	81.54
German	82.98	$>$ (99%)	81.61	$>$ (96%)	81.16
Czech	70.06	$>$ (99%)	69.10	$<$ (90%)	69.56
English	87.32	$>$ (99%)	86.77	$<$ (97%)	87.14
Average	*81.02*		*80.46*		*80.29*

Table 4.7: Summary of table 4.8 containing only the test accuracy after training with the largest data sets. The "Sign" columns shows the statistical significance of the difference between the two tree methods and the simple division strategy. For example the statistical certainty that the simple division strategy gives better accuracy than the tree division strategy with intuitive division order for a Swedish data set of the size used in the experiment is greater than 19%.

amount of memory which causes the memory usage to 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 Decision 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 Intuitive	**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 Gain Ratio	**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 Intuitive	**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 Gain Ratio	**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 Intuitive	**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 Gain Ratio	**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 Intuitive	**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 Gain Ratio	**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 Intuitive	**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 Gain Ratio	**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: The table contains 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** = *Labeled 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 improve 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 four main types to chose from, namely LIBLINEAR, LIBSVM alone or combined with a division strategy that divide the training data on one feature. 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.4.

The decision tree can be created either by manually configuring 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 experiment described in section 4.7 shows it is not obvious which of the alternatives is best to use. 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, but there were indications that the Gain Ratio calculated division order might give better results if more advanced feature extraction models could be used.

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.

Different values of the parameters show that it is difficult to give 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 better to use a low number of cross validation divisions because it 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 were 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 project 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 limitations and interesting future work.

Goal number 1 was to investigate how division of the training data effects the performance of the resulting dependency parsing system. This has been investigated 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 division approach could easily be scaled out to several processors 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 on the accuracy when nonlinear SVMs are used. To summarize the findings about how the accuracy is affected 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 faster than linearly with the number of training examples for nonlinear SVMs.

Goal number 2 was to investigate the theoretical reasons for the effect of division. The hypothesis that has been discussed in this report and which many of the experiments in chapter 4 support, is that when division on certain features is 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 was 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 a greater positive effect on the accuracy if the training set is larger. The training sets available for different languages will probably be larger in the future 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 one of the experiments. To get 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 indicates that an improvement could be gained even with such parameters the amount of RAM memory needed was a practical limitation for doing such experiments. A suggestion for how to fix this is presented in section 6.2.

There are a lot additional experiments that would be interesting to perform that just could not be done because of the time limitation of the projects. For example it would have been interesting to perform experiments with different algorithms for the decision tree creation.

6.2 Future work

All the experiments that have been conducted during the thesis work in a real dependency parsing setting have used non-optimal feature extraction models and parameters. The reasons for that has 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 greatest obstacle for achieving 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 becomes too long. A better solution would be to have the models loaded on several different computers. This could have the additional benefit of making it possible to easily speed 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 division 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 could 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. In *CoNLL-X '06 Proceedings of the Tenth Conference on Computational Natural Language Learning*, 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, 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.

[GE08] Y. Goldberg and M. Elhadad. splitSVM: fast, space-efficient, non-heuristic, polynomial kernel computation for NLP applications. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, pages 237–240. Association for Computational Linguistics, 2008.

[HCJ+09] Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia, Martí Lluís, Màrquez, Adam Meyers, Joakim Nivre, and Sebastian Padó. The CoNLL-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *CoNLL '09 Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task*, 2009.

[HPH+] 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. CD-rom. ISBN:1-58563-370-4.

[KMN09] S. Kübler, R. McDonald, and J. Nivre. *Dependency parsing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 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.

[NHK+07] Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL*, pages 915–932, 2007.

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

[NM07] J. Hall J. Nilsson A. Chanev G. Eryigit S. Kübler S. Marinov Nivre, J. and E. Marsi. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135, 2007.

[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, 2006.

[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 International Workshop on Parsing Technologies (IWPT 2003)*, volume 3, 2003.

Appendix A

Experiment Diagrams

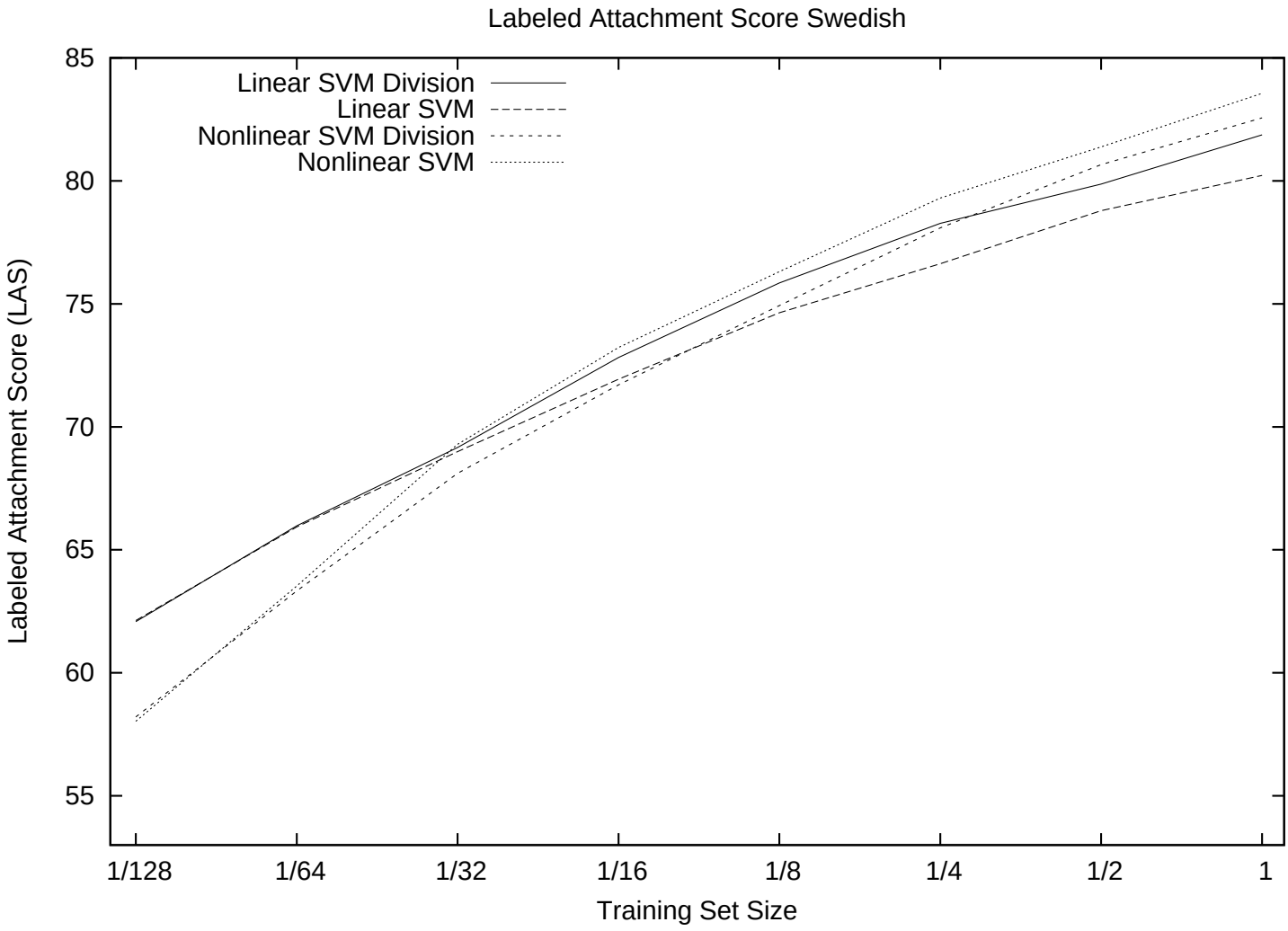

Figure A.1: The digram visualizes the Labeled Attachment Score for all training set sizes created from the Swedish data set and tested in the experiment presented in section 4.1. The values used to create the diagram can be seen in table 4.1 and 4.2. Please, note that the x-axis in the diagram has logarithmic scale.

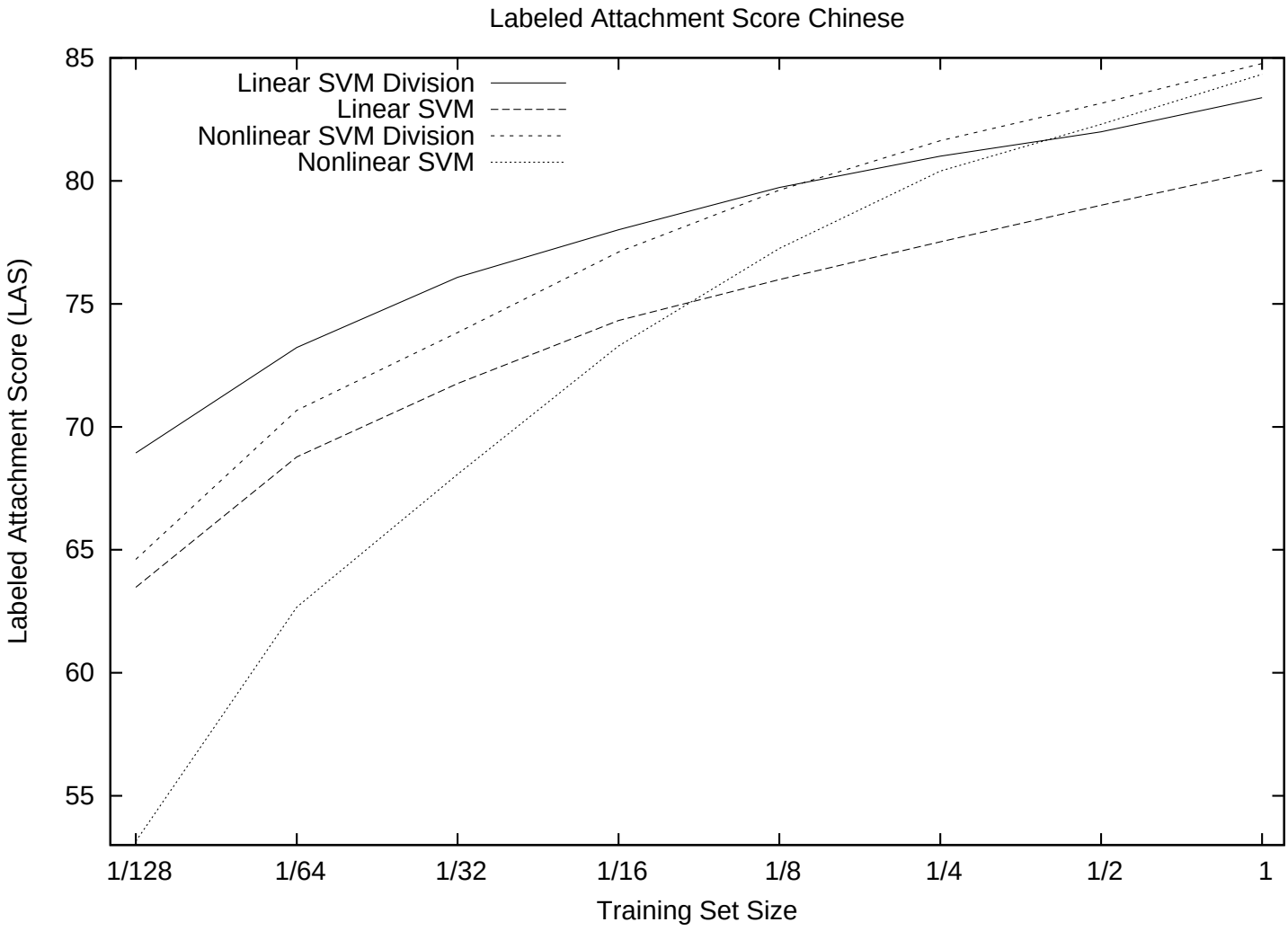

Figure A.2: The digram visualizes the Labeled Attachment Score for all training set sizes created from the Chinese data set and tested in the experiment presented in section 4.1. The values used to create the diagram can be seen in table 4.1 and 4.2. Please, note that the x-axis in the diagram has logarithmic scale.

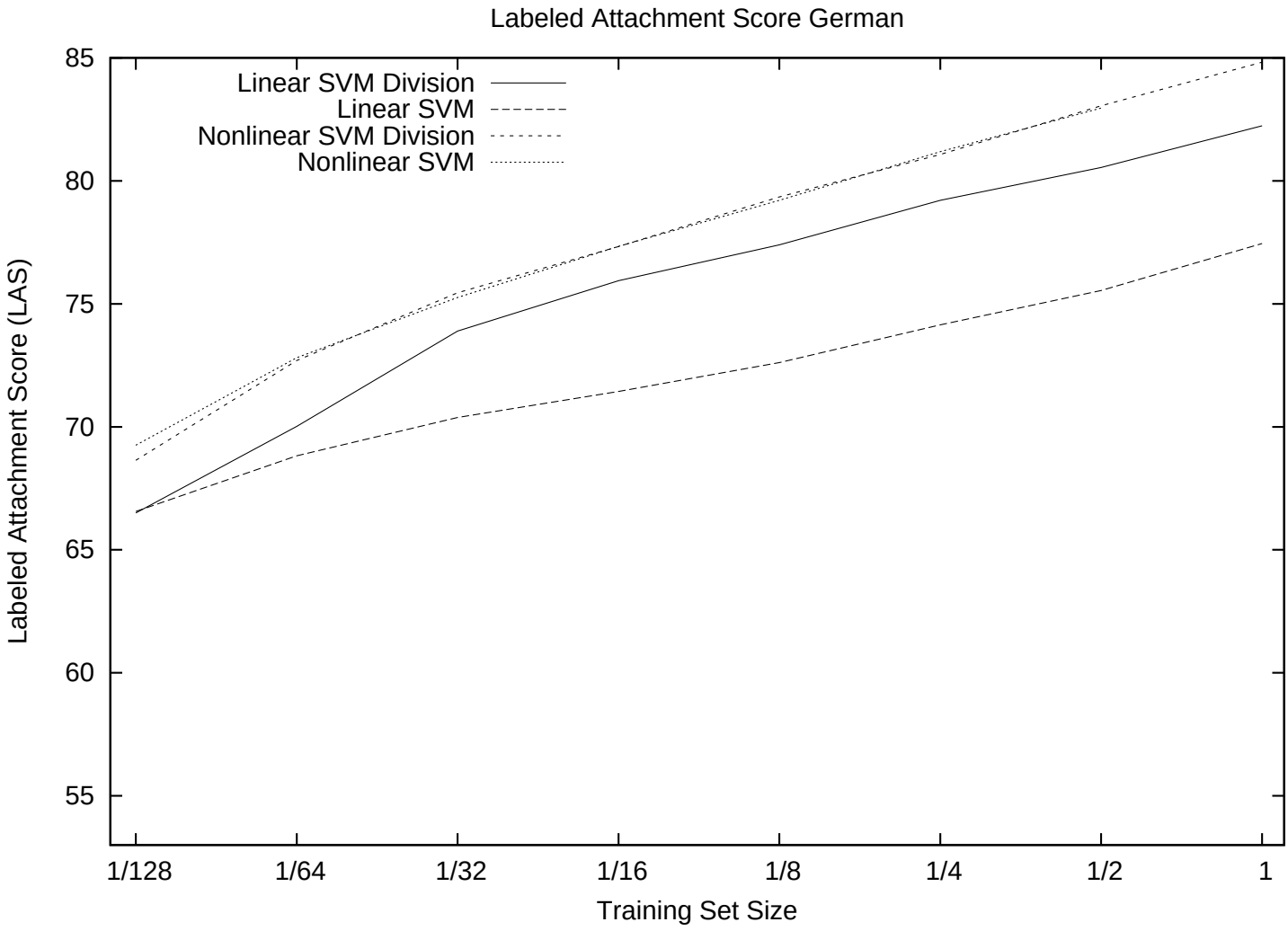

Figure A.3: The digram visualizes the Labeled Attachment Score for all training set sizes created from the German data set and tested in the experiment presented in section 4.1. The values used to create the diagram can be seen in table 4.1 and 4.2. Please, note that the x-axis in the diagram has logarithmic scale.

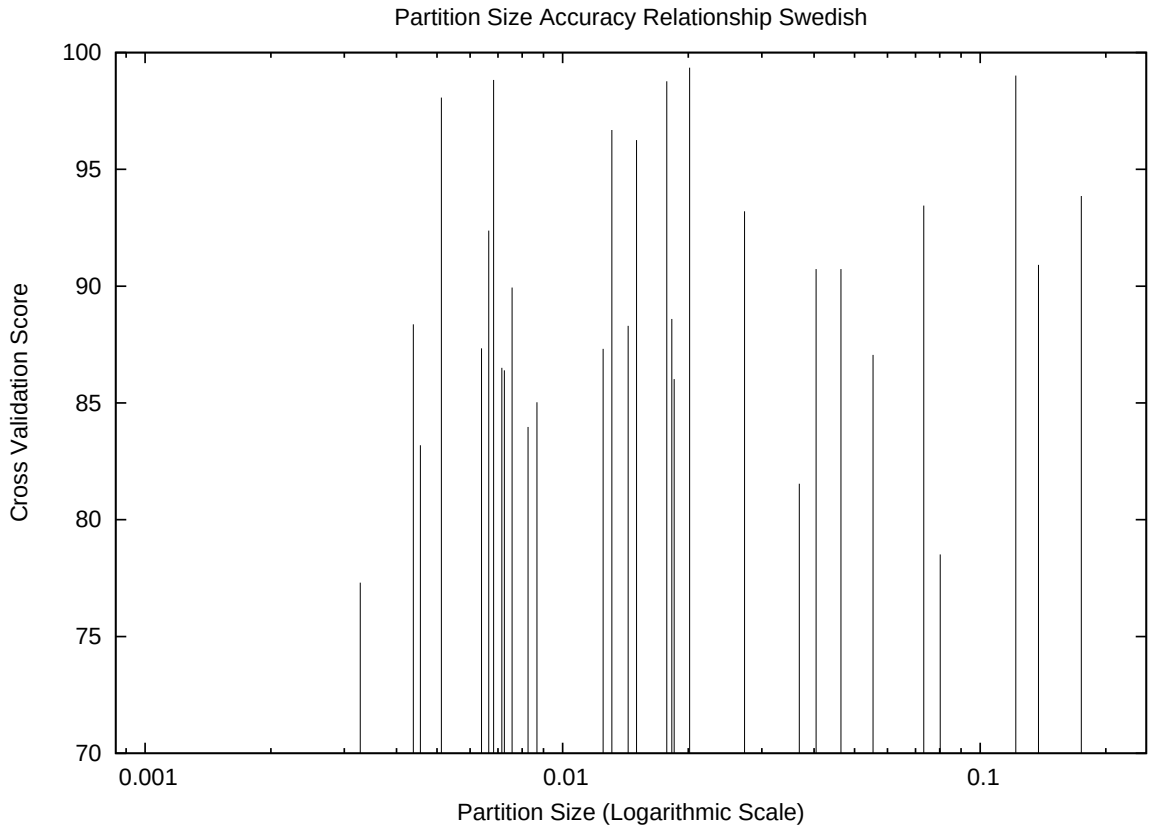

Figure A.4: The diagram illustrates that there is no clear pattern between size of partitions created when dividing the training data and the cross validation accuracy. Every spike in the diagram represents a partition created when dividing Swedish. The partition size in the diagram is the fraction of the total training set size.

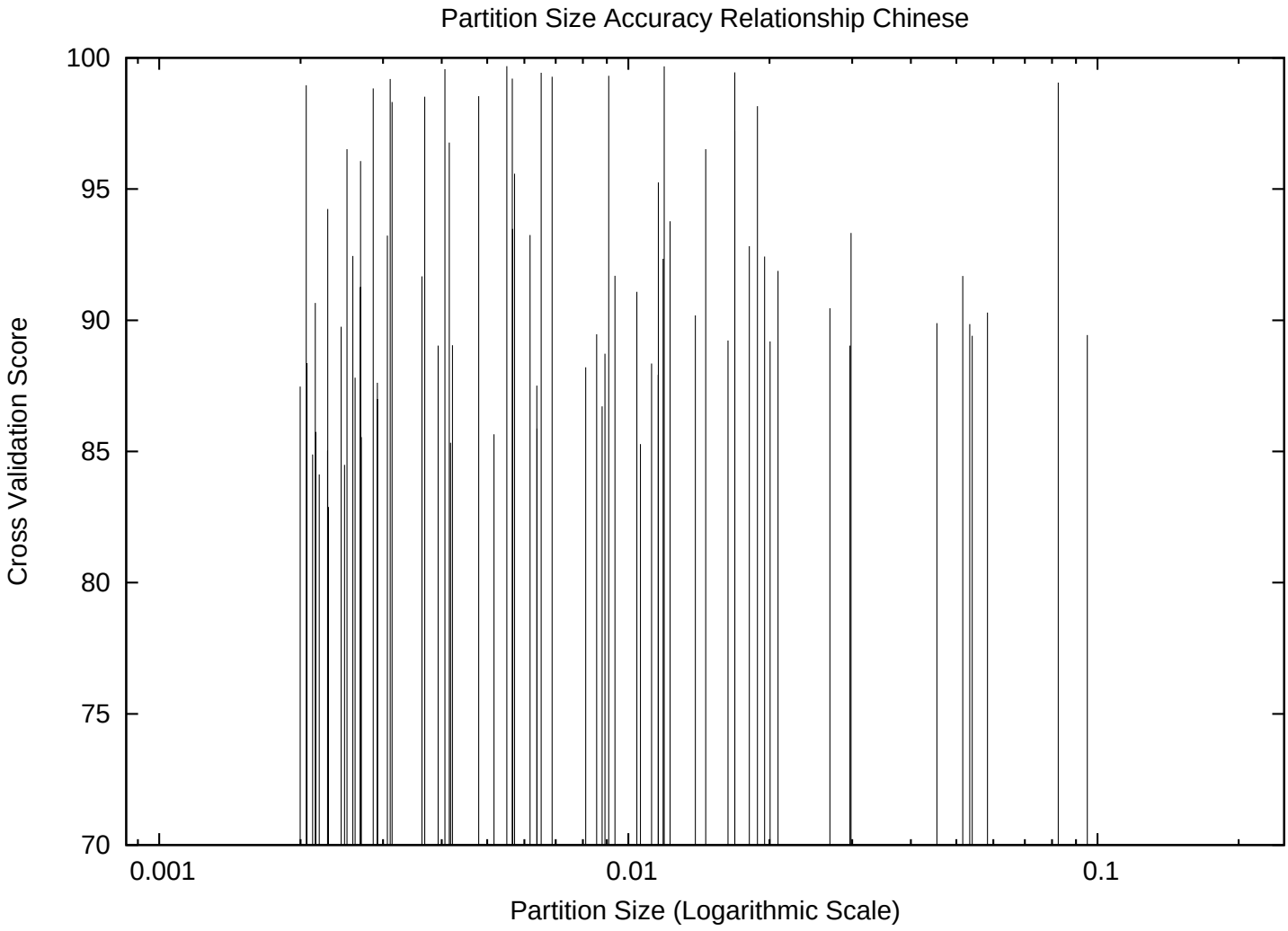

Figure A.5: The diagram illustrates that there is no clear pattern between size of partitions created when dividing the training data and the cross validation accuracy. Every spike in the diagram represents a partition created when dividing Chinese. The partition size in the diagram is the fraction of the total training set size.

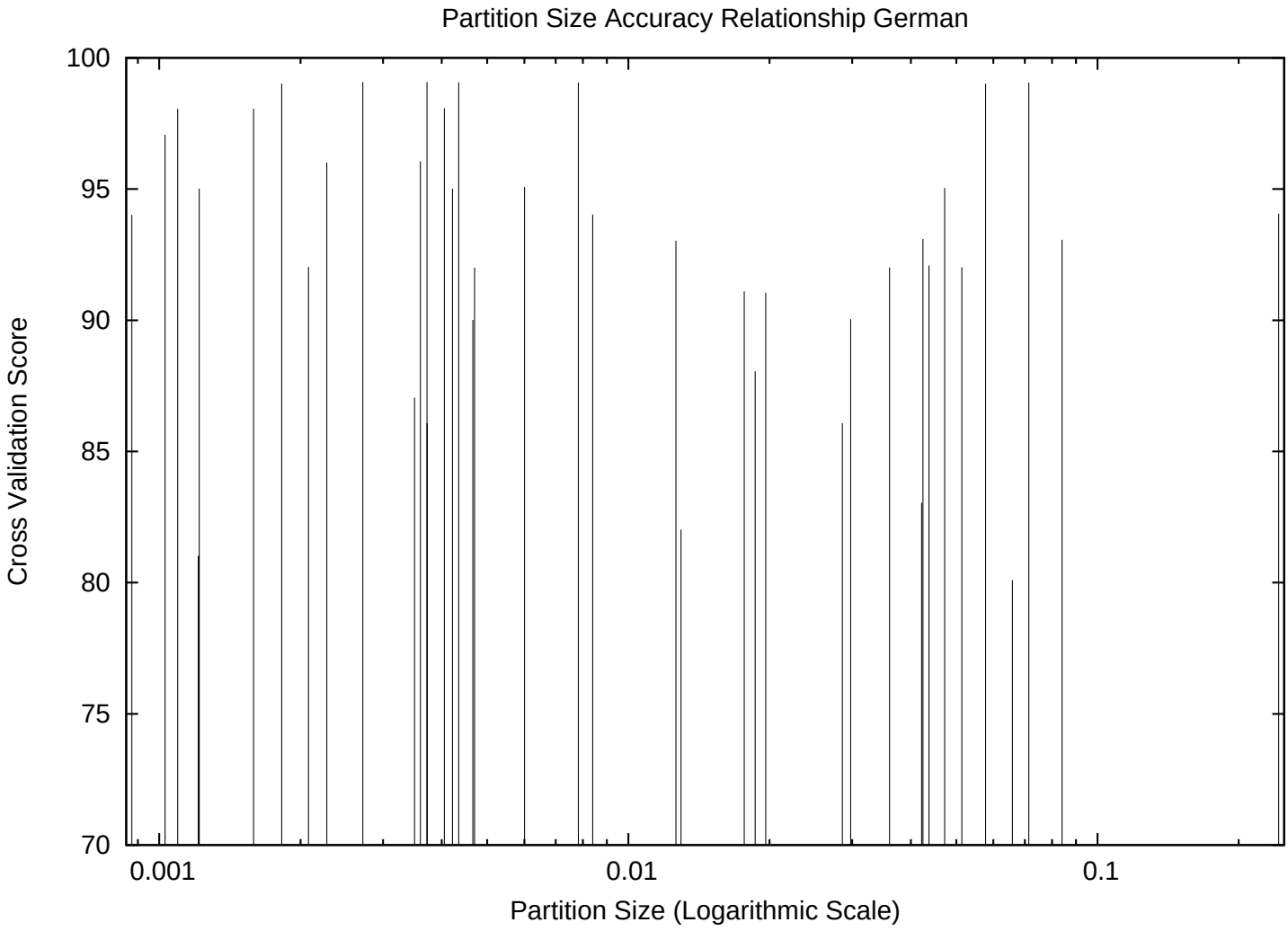

Figure A.6: The diagram illustrates that there is no clear pattern between size of partitions created when dividing the training data and the cross validation accuracy. Every spike in the diagram represents a partition created when dividing German. The partition size in the diagram is the fraction of the total training set size.

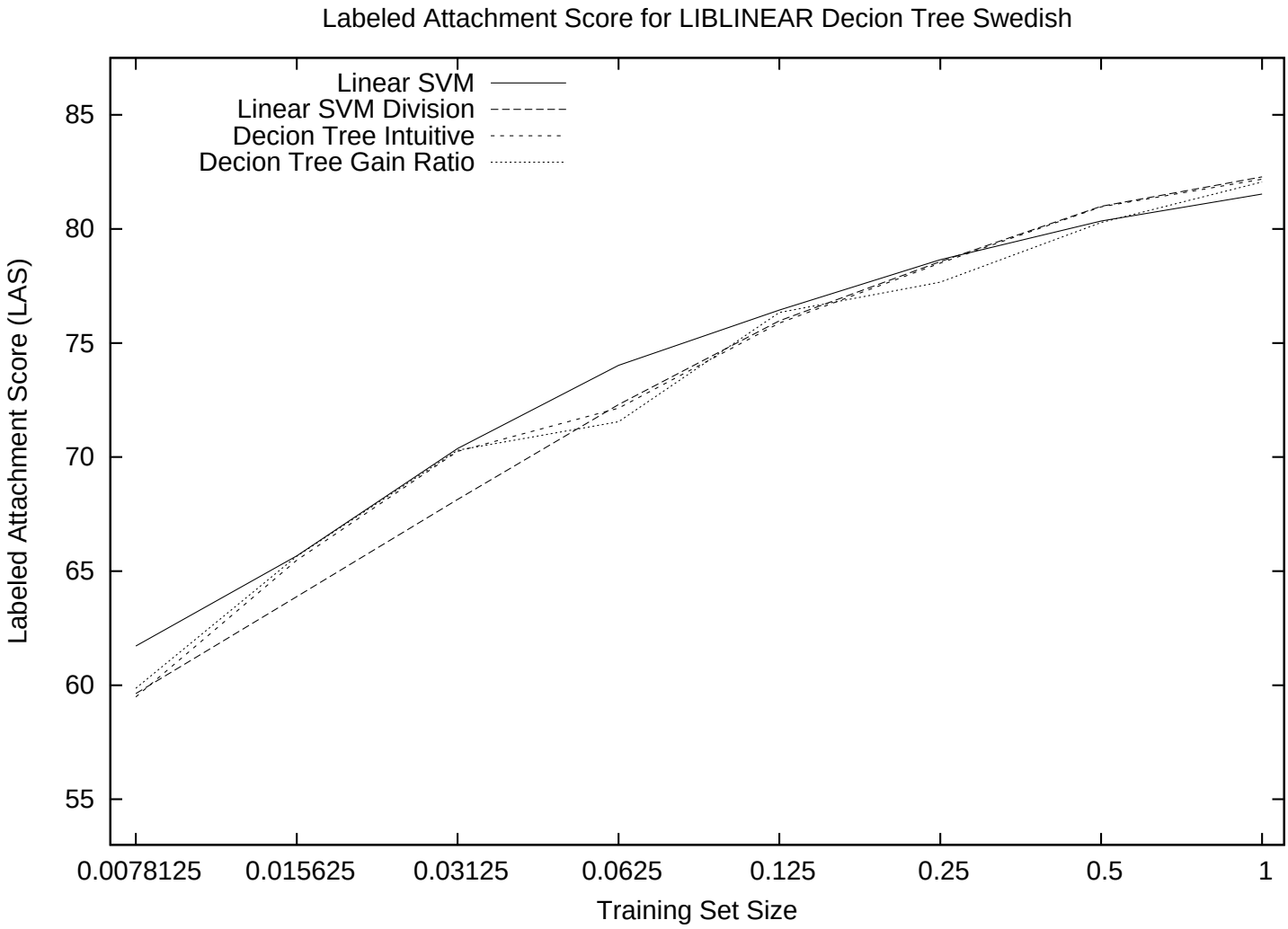

Figure A.7: The digram visualizes the Labeled Attachment Score for all training set sizes created from the Swedish data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.8. Please, note that the x-axis in the diagram has logarithmic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

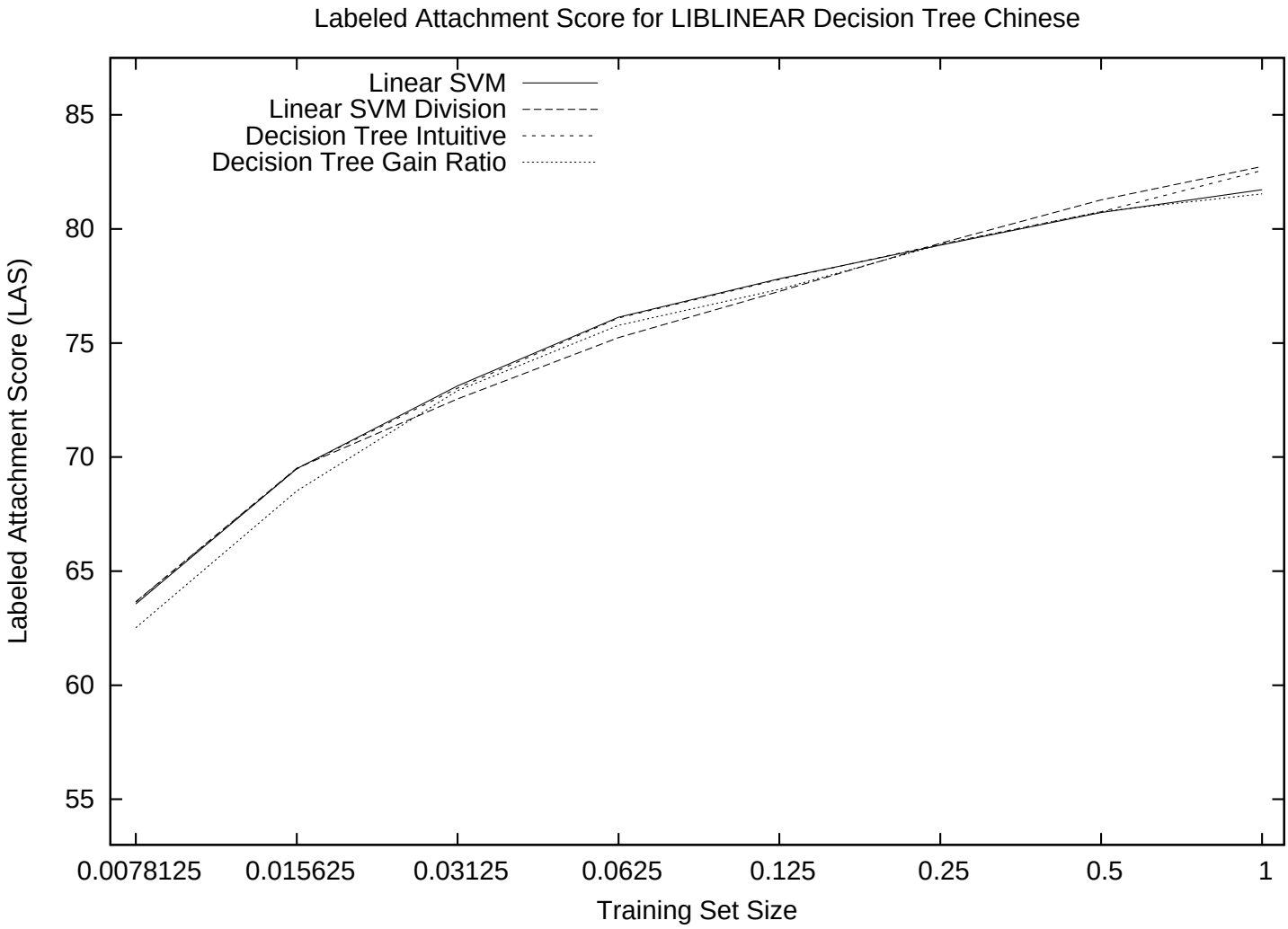

Figure A.8: The digram visualizes the Labeled Attachment Score for all training set sizes created from the Chinese data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.8. Please, note that the x-axis in the diagram has logarithmic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

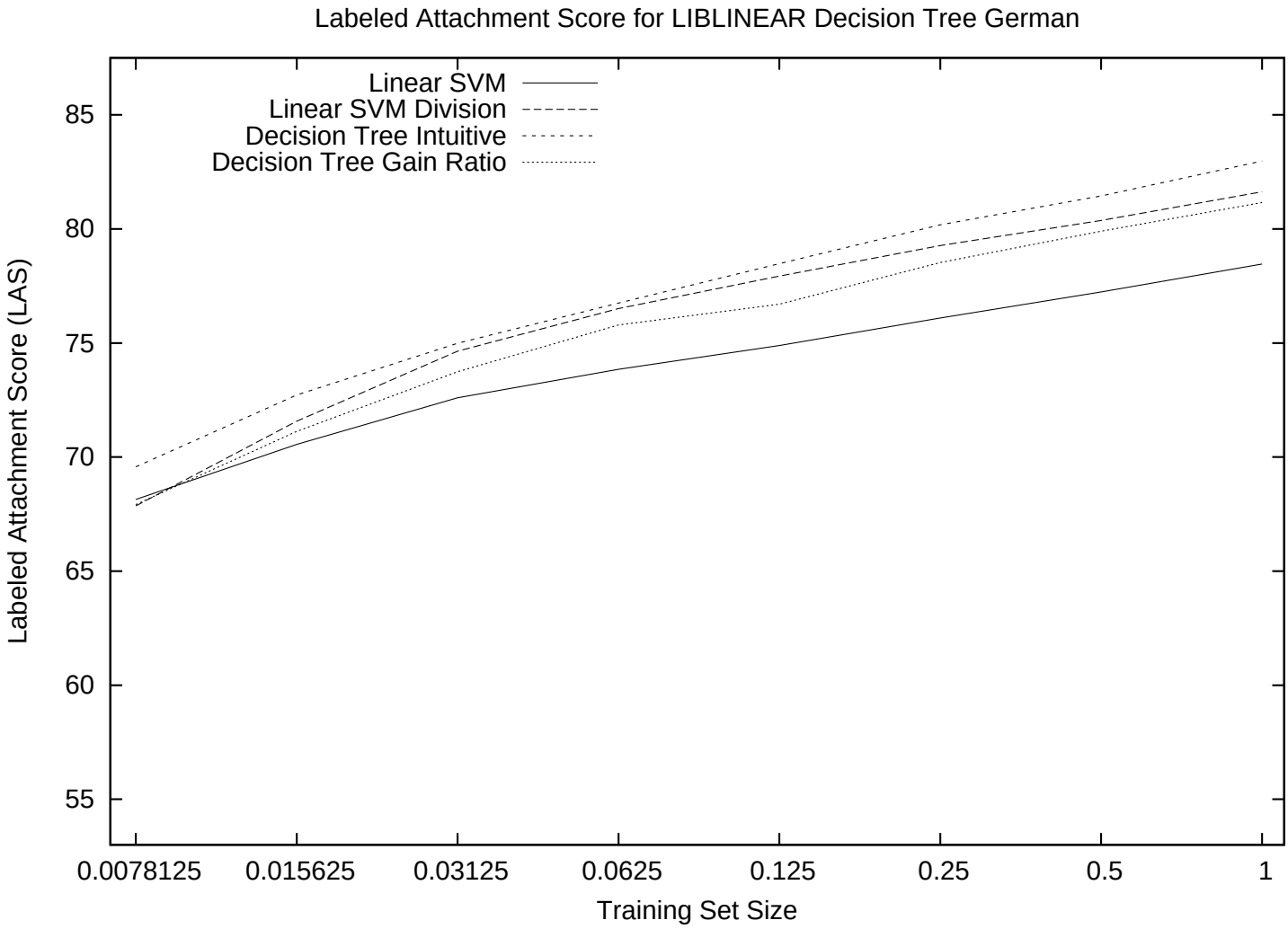

Figure A.9: The digram visualizes the Labeled Attachment Score for all training set sizes created from the German data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.8. Please, note that the x-axis in the diagram has logarithmic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

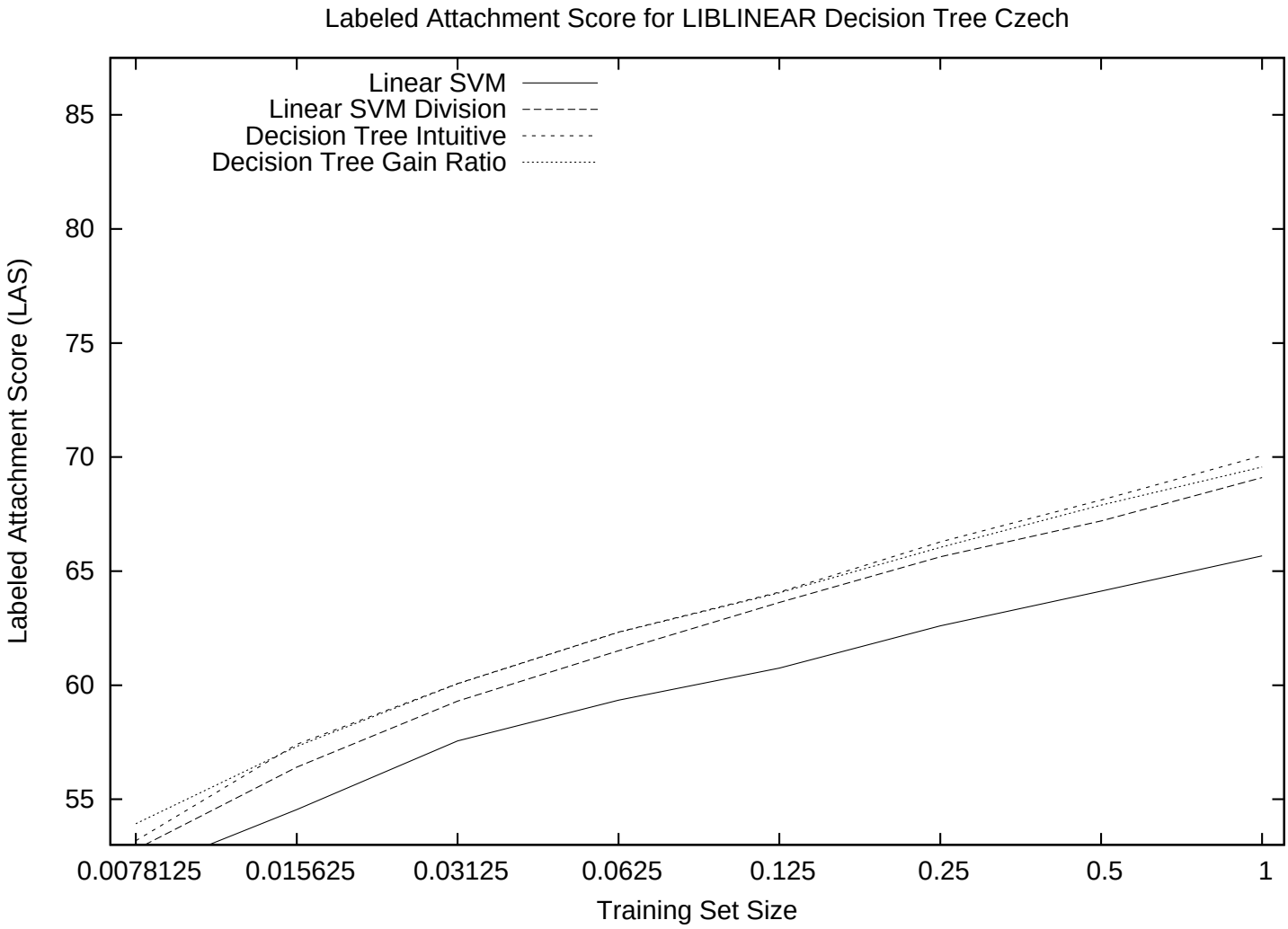

Figure A.10: The digram visualizes the Labeled Attachment Score for all training set sizes created from the Czech data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.8. Please, note that the x-axis in the diagram has logarithmic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

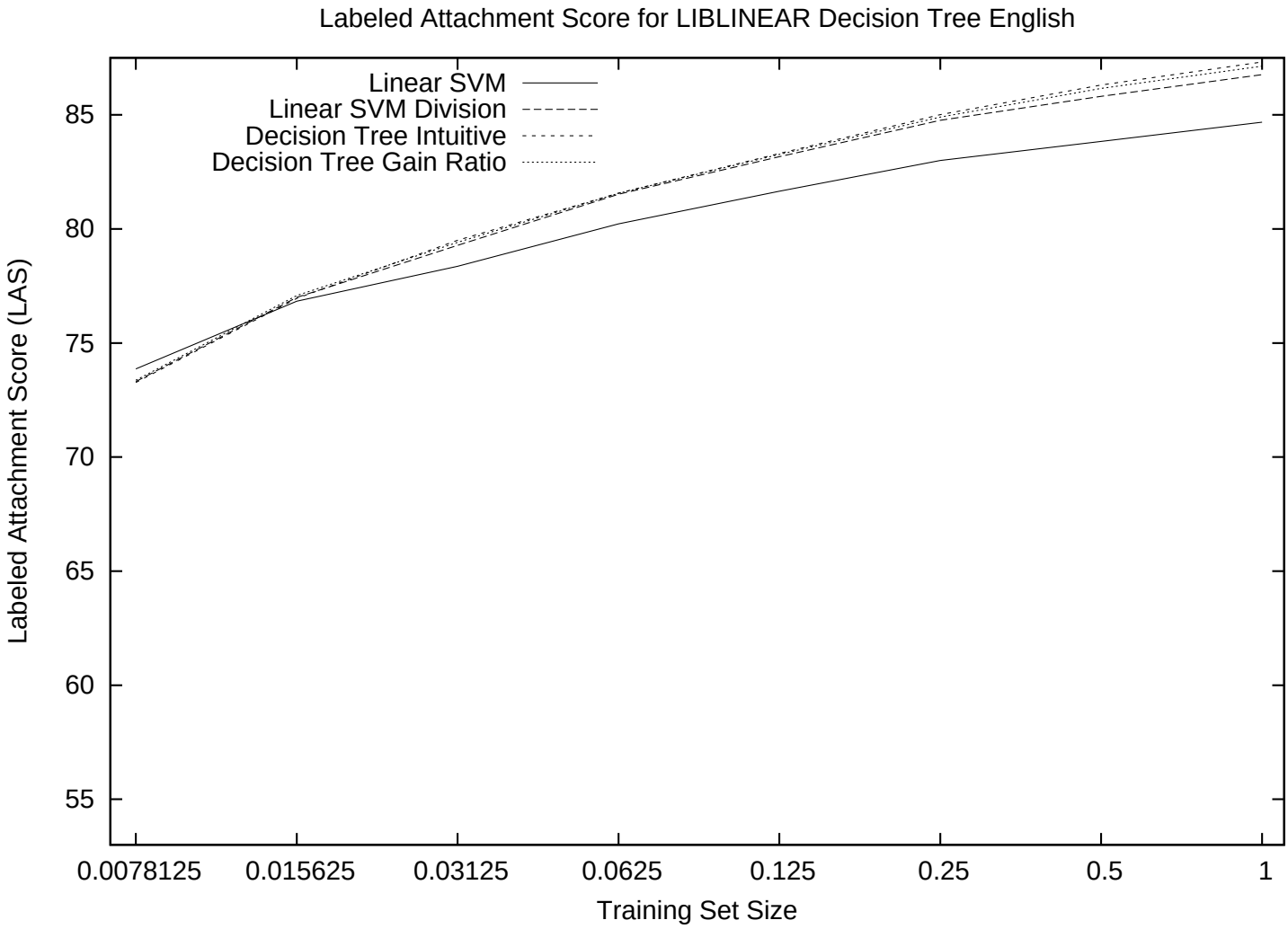

Figure A.11: The digram visualizes the Labeled Attachment Score for all training set sizes created from the English data set and tested in the experiment presented in section 4.7. The values used to create the diagram can be seen in table 4.8. Please, note that the x-axis in the diagram has logarithmic 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 the Nivre-Arc-eager parsing algorithm shall be 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 that will be used as feature extraction model. The feature extraction configuration used in the experiments presented in chapter 4 is 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 this section 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. The feature extraction models presented in the following sections have been used in the experiments presented in section 4.4, 4.5 and 4.6.

B.2.1 English Stack Projective

```
<?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 Projective

```
<?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 LIBLINAR and LIBSVM settings

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.

B.4 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 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="50"/>
```

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"/>
```