Performance-Oriented Dependency Parsing

Dissertation zur Erlangung des Grades eines Doktors der Philosophie der Philosophischen Fakultäten I und II der Universität des Saarlandes

eingereicht von:

Alexander Volokh

Contents

List of Figures

List of Tables

Abstract

Dependency parsing has become very popular among researchers from all NLP areas, because dependency representations contain very valuable easy-to-use information. In the last decade a lot of dependency parsers have been developed, each of them somehow special with its own unique characteristics. In the course of this thesis I have developed yet another parser - MDParser. In this work I discuss the main properties of dependency parsers and motivate MDParser's development. I present the state of the art in the field of dependency parsing and discuss the shortcomings of the current developments. To my mind the main problem of the current parsers is that the task of dependency parsing is treated independently of what happens before and after it. Therefore the preprocessing steps and the embedding in applications are neglected. However, in practice parsing is rarely done for the sake of parsing itself, but rather in order to use the results in a follow-up application. Additionally, current parsers are accuracy-oriented and focus only on the quality of the results, neglecting other important properties, especially efficiency. The design of MDParser tries to counter all these drawbacks.

The evaluation of some NLP technologies is sometimes as difficult as the task itself. For dependency parsing it was long thought not to be the case, however, some recent works show that the current evaluation possibilities are limited. In this thesis I broadly present and discuss both intrinsic and extrinsic evaluation methodologies for dependency parsing. Both approaches have numerous disadvantages which I demonstrate in my work. The attachment scores, which are the most used metric of the intrinsic evaluation, do not differentiate between different dependency types, are being computed for the same portions of treebanks since many years, and thus often promote overfitting to this particular kind of data, which is especially dangerous because the data contains a certain amount of inconsistencies. The extrinsic evaluation, which evaluates the contribution of parser results to a certain application is also problematic, because the embedding of parsing into applications requires a lot of expertise and implementational effort and it is unclear whether the impact on the result is due to the quality of the results or of the embedding. In the thesis I propose a methodology to account for those weaknesses and combine the strengths of both evaluation methodologies.

Finally, I evaluate MDParser and compare it with other state-of-the-art parsers. The results show that I was able to create the fastest parser currently

available, which is able to process plain text, which other parsers usually can not and whose results are only slightly behind the top accuracies in the field. However, I analyse the gap as well as its impact on applications and demonstrate that it is not decisive.

This thesis contains the descriptions of many experiments which I have performed in the course of the last three years in order to achieve the goal of implementing and evaluating MDParser. Many of these experiments were unsuccessful or their results have become obsolete in the course of my work. However, both types of work were essential for achieving the final results, because I have learned a lot from most of the experiments I have done, independently of their success.

One of the central parts of any data-driven parser is its machine learning component. I have had to adopt MDParser to four different machine learning approaches in the course of the time in order to keep up with the current developments. I have started with maximum entropy classification. The first package I have used contained only a very simple training technique, so I had to replace it soon after I was not able to achieve satisfactory results. The second package contained the best possible training method for maximum entropy classifiers, however, the result still was not satisfying. The main problem was that for linear classification a lot of features are required and if one is not able to select good ones for training, then the training is very expensive. Thus I have given up maximum entropy classification and looked for a linear classifier which supported feature selection. The third attempt was a linear regression classifier with regularisation. In regularisation a penalty term is applied to every feature weight, so that only the features with a big weight remain until the end of the learning. This corresponds to an implicit feature selection, since many bad features can not surpass the penalty term and only good ones remain. However, the number of instances was still a problem. That is why I have finally switched to linear support vector machines. With support vector machines not all feature vectors are of equal importance. Only so called support vectors, i.e. instances of data which are involved into drawing the margin between the classes, are relevant and weights only for their features are learned. These numerous changes always required significant code modifications in the system and a lot of work had to be done from scratch. However, I was able to better understand the significance and influence of machine learning on the parser in general, independently of a specific machine learning approach.

Machine learning for data-driven parsing is usually a very expensive task

from the computational point of view, because there are millions of instances with millions of features. Even though I have always used linear classification approaches, i.e. there was no need to explore the combinations of individual features, it has always been a lengthy process which required sophisticated hardware. Therefore I have invested a lot of effort in reducing the complexity of the problem. On the one hand I have tried to split the problem into subproblems, whose individual solutions are not as costly as the entire problem. On the other hand I have tried to reduce the number of features by preselecting the most useful ones before training a model. Both approaches have become obsolete with the final machine learning approach which is used in MDParser. The final strategy which is based on support vector machines contains an implicit feature selection mechanism and can work with the entire set of features without any need for prefiltering. Since support vector machines only require support vectors, which are a small fraction of all vectors, the computation is no longer expensive.

One of the most important achievements of this thesis is that I show that the efficiency of dependency parsing is not mainly dependent on the complexity of the underlying parsing strategy, i.e. the worst-case number of necessary states when a processing a sentence. Currently people prefer linear strategies, for instance Nivre's algorithms over more complex quadratic ones, as for example Covington's parsing strategy, without analysing other properties. I have used profiling technology in order to compute the exact amount of processing time for every step and component of the algorithm. It turns out that the overwhelming amount of time is spent on feature extraction, namely on string concatenations. However, when processing a sentence with some parsing strategy, the feature extraction is not always necessary in every state and therefore it is not enough to simply compare the number of states necessary for different strategies, but one has to rather compare the number of the actually required feature extractions. This number is similar for both Nivre's and Covington's parsing strategies, even if they have a different theoretical complexity. In addition to that, techniques like memoisation, which allow reusability of features, such that expensive string operations do not have to be unnecessarily repeated, can be applied easier to Covington's algorithm rather then to Nivre's linear ones and in the end it can even outperform them.

One more very successful piece of work included the experiments on the treebanks. Treebanks are supposed to be of a very high quality, because they are essential for the success of all data-driven approaches which learn from

them. I was able to show that it is in fact not the case and treebanks contain a considerable amount of erroneous annotation. Thus some more sophisticated approaches have higher scores partially because they are better at replicating these inconsistencies and thus despite the scores their results are objectively not of that much higher quality. Moreover, the approach I have come up is able not only to detect inconsistencies, but also corrects them.

Furthermore, treebanks for some languages are very big and learning curve experiments show that almost the same result can already be achieved with less amount of data. For other languages the treebanks are too small and much better results could be achieved if more annotated data were available. Since annotation of corpora is an expensive process, it is important to control it as good as possible. A very interesting question here, which I have successfully investigated in my work, is what type of data exactly is useful and should be annotated first, because it is both more beneficial for the performance and requires less effort to be annotated.

Finally, I have used various opportunities to participate in shared tasks for recognising textual entailment in order to apply my parser to concrete tasks. The first task called PETE was specially designed for parser evaluation and provided an interesting data set, which required the parser to correctly recognise a limited set of different dependencies. The other two tasks RTE-6 and RTE-7 were not intended to be used for parser evaluation, however, I have used their data sets for testing parser performance on real-world data. This work allowed me to better understand how the embedding of dependency parsing into applications works. Furthermore, these experiments demonstrate that the performance of parsers for the standard development data of treebanks is different to that for real world data sets. I have already stated that a big shortcoming of the currently predominant evaluation is that the performance of parsers is averaged over all dependency types. Some recent literature tries to address this fact by limiting the evaluation to a subset of dependency types, which are of a particular importance for some task. I have performed a similar analysis for the task of RTE and present my findings in this thesis.

A lot of the work presented in this thesis has been published on various conferences. Thus besides a well-working dependency parser, there are also seven papers, which have been peer-reviewed and accepted on different international conferences, which arise from this work.

Zusammenfassung

Dependenzparsing steht unter Wissenschaftlern aus verschiedenen Gebieten der Verarbeitung natürlicher Sprache hoch im Kurs, weil Dependenzstrukturen sehr wertvolle Informationen enthalten, die leicht verwendet werden können. Im den letzten zehn Jahren wurden viele Dependenzparser entwickelt, jeder von ihnen mit speziellen Eigenschaften, die ihn von anderen Parsern absetzen. Im Zuge der Arbeit an dieser Dissertation habe ich einen weiteren Dependenzparser namens MDParser entwickelt. In dieser Arbeit erörtere ich die wichtigsten Eigenschaften von Dependenzparsern und motiviere die Entwicklung meines Parsers. Ich präsentiere den derzeitigen Stand der Wissenschaft auf dem Gebiet des Dependenzparsing und diskutiere über die Nachteile aktueller Entwicklungen. Meiner Meinung nach liegt das größte Problem aktueller Parser darin, dass sie die Parsingaufgabe unabhängig von vorausgehenden oder nachfolgenden Verarbeitungsschritten bzw. Anforderungen behandeln. Aus diesem Grund werden die Vorverarbeitungsschritte und das Integrieren der Parser in die Anwendungen vernachlässigt. In der Praxis jedoch ist Parsing selten eine isolierte Aufgabe, sondern liefert Ergebnisse für nachfolgende Anwendungen. Außerdem konzentrieren sich aktuelle Parser nur auf die Qualität der Ergebnisse in Bezug auf bestimmte Mengen von Trainings- und Evaluationsdaten. Sie vernachlässigen dabei andere wichtige Eigenschaften, insbesondere Effizienz. Das Design des MDParsers versucht all diesen Nachteilen entgegenzuwirken.

Die Evaluierung von Technologien in der Verarbeitung natürlicher Sprache ist manchmal genauso kompliziert wie die Lösung der Aufgabenstellung selbst. Für Dependenzparsing schien dies lange Zeit nicht so zu sein, jedoch zeigen einige neuere Arbeiten, dass die derzeitigen Evaluationsmöglichkeiten starke Einschränkungen aufweisen. In dieser Arbeit präsentiere und erörtere ich sowohl intrinsische als auch die extrinsische Evaluierungsmethoden für Dependenzparsing. Beide Ansätze haben zahlreiche Nachteile, die ich in dieser Arbeit demonstriere. *Attachment Scores* sind die am häufigsten benutzte Metrik der intrinsischen Evaluation. Ihr Nachteil besteht darin, dass sie nicht zwischen unterschiedlichen Dependenztypen unterscheiden kann und seit Jahren für dieselben Teilmengen einiger Baumbanken berechnet wurde. Auf diese Weise wurde eine Überanpassung an diese Art von Daten gefördert, was insbesondere deswegen problematisch ist, weil diese Daten nicht zu vernachlässigende Fehler enthalten. Die extrinsische Evaluierung, die den Beitrag der Ergebnisse eines Parsers zur Lösung einer Aufgabe bewertet, ist problematisch, weil die Integration von

Parsern in Anwendungen hohe Sachkompetenz und Implementierungsaufwand erfordert und viele Faktoren einen Einfluss auf die Qualität des Endergebnis haben, nicht nur die Qualität der Ergebnisse, sondern auch die Qualität der Integration bzw. Anpassung der Ergebnisse an die Anforderungen der Anwendung. In dieser Arbeit schlage ich eine Methode vor, die diese Schwächen berücksichtigt und die Stärken beider Methoden kombiniert.

Die Evaluation meines MDParsers schließt natürlich den Vergleich mit anderen Parsern ein. Die Ergebnisse zeigen, dass der MDParser zur Zeit die schnellste verfügbare Implementierung eines Dependenzparsers darstellt, der in der Lage ist, reinen Text zu verarbeiten, eine Eigenschaft, die nicht viele Dependenzparser vorhalten. Die Evaluationsergebnisse liegen nur minimal hinter anderen Top-Ergebnissen zurück. Ich habe diese Differenz und ihren Einfluss auf Anwendungen untersucht und gezeigt, dass sie für die Erfüllung spezieller Aufgaben nicht entscheidend ist.

In dieser Arbeit beschreibe ich viele Experimente, die ich im Verlauf der letzten drei Jahre durchgeführt habe, mit dem Ziel, MDParser zu implementieren und zu evaluieren. Viele dieser Experimente waren erfolglos oder ihre Ergebnisse sind im Laufe meiner Arbeit überholt worden, weil sehr viele andere Gruppen ständig neuere Arbeiten zum Dependenzparsing veröffentlichen. Aber auch diese Experimente waren unerlässlich, weil ich aus ihrer Durchführung, unabhängig von ihrem Erfolg, sehr viel gelernt habe, und damit zu den schlussendlichen Ergebnissen dieser Arbeit beigetragen haben.

Ein zentraler Teil eines datenbasierten Dependenzparsers ist die Komponente zum maschinellen Lernen (ML) der statistischen Datenmodelle. Ich musste im Verlauf der Zeit MDParser an vier unterschiedliche Ansätze anpassen, um mit aktuellen Entwicklungen Schritt zu halten. Ich habe mit einer ML-Softwarebibliothek begonnen, die auf der Maximum-Entropie-Methode basiert. Die von dieser Bibliothek bereitgestellte, sehr elementare Trainingsmethode, konnte nur sehr mäßige Ergebnisse erzielen, so dass sie kurz darauf ersetzt werden musste. Die zweite von mir benutzte Bibliothek implementiert zwar die bestmögliche Trainingsmethode für die Maximum-Entropie-Methode, lieferte aber trotzdem keine zufriedenstellenden Ergebnisse. Das größte Problem bestand darin, dass für die lineare Klassifikation sehr viele Features notwendig sind. Wenn man nicht in der Lage ist nur die besten von ihnen für das Training auszuwählen, dann ist das Training sehr berechnungsintensiv. Aus diesem Grund gab ich die Maximum- Entropie-Methode zugunsten eines linearen Klassifikator auf, der die Featureauswahl unterstützt. Im dritten Versuch habe ich einen linearen

Klassifikator mit Regularisierung genommen. Bei der Regularisierung wird ein Strafterm auf jedes Featuregewicht angewendet, so dass nur die Features mit einem sehr großen Gewicht bis zum Ende des Trainings übrig bleiben. Dies entspricht einer impliziten Featureauswahl, da schlechte Features durch den Strafterm aussortiert werden und nur die guten übrig bleiben. Jedoch führt auch bei dieser Art von Klassifikator die sehr große Anzahl an Trainingsdaten immer noch zu Problemen. Aus diesem Grund habe ich schließlich lineare Support-Vektor-Maschinen (SVM) eingesetzt. Bei Support-Vektor-Maschinen sind nicht alle Featurevektoren von gleicher Bedeutung. Nur die so genannten Support-Vektoren, d.h. Trainingsinstanzen die relevant für die Abgrenzung zwischen unterschiedlichen Klassen sind, werden benutzt und nur für diese Featurevektoren werden Gewichte gelernt. Der mehrfache Wechsel der ML-Module erforderte stets weitreichende Änderungen im Programmcode des Systems; vieles musste von Grund auf neu implementiert werden. Allerdings verdeutlichen die auftretenden Probleme die generelle Bedeutung und den Einfluss des maschinellen Lernens auf den Parser, unabhängig von der verwendeten Methode.

Maschinelles Lernen für datenbasiertes Parsing bedeutet in der Regel einen immensen Rechenaufwand, weil es Millionen von Trainingsinstanzen mit Millionen von Features gibt. Obwohl ich immer lineare Klassifizierungsmethoden benutzt habe, so dass es für die Klassifikatoren nicht notwendig war auch Kombinationen von einzelnen Features zu betrachten, waren die Trainingsläufe bei hohen Hardwareanforderungen trotzdem immer langwierig. Aus diesem Grund habe ich viel Aufwand betrieben um die Komplexität des Problems zu reduzieren. Zum einen habe ich versucht das Problem in kleinere Teilprobleme aufzuteilen, deren einzelne Lösungen weniger kostspielig sind als das Gesamtproblem. Zum anderen habe ich versucht die Anzahl der Features zu reduzieren, indem ich die nützlichsten bereits vor dem Trainieren eines Modells ausgewählt habe. Durch die endgültige Wahl von Support-Vektor Maschinen (SVM) zum Lernen der Modelle wurden diese Anstrengungen aus zwei Gründen obsolet. Zum einen enthält die Trainingsphase von SVM einen impliziten Featureauswahlmechanismus, so dass keine manuelle Vorauswahl nötig ist, zum anderen selektieren SVM aus den Trainingsinstanzen die besten Support-Vektoren, die nur noch einen Bruchteil der ursprünglichen Menge ausmachen, was den Rechenaufwand stark reduziert.

Eines der wichtigsten Ergebnisse der Arbeit ist, dass ich zeige, dass die Effizienz des Dependenzparsing nicht überwiegend von der Komplexität der zu Grunde liegenden Parsingstrategie abhängt, die durch die worst-case Anzahl

der benötigten Zustände für die Verarbeitung eines Satzes bestimmt ist. Zur Zeit werden lineare Strategien, wie die Algorithmen von Nivre, vor komplexeren quadratischen, wie z.B. der Parsingstrategie von Covington, bevorzugt. Dabei wird ignoriert, dass der Worst-case nur selten eintritt und die benötigte Rechenzeit von tieferliegenden Rechenprozessen dominiert werden kann. Mittels Profiling habe ich die exakte Rechenzeit bestimmt, die für jeden Schritt und jede Komponente des Parsingalgorithmus notwendig ist. Es stellte sich heraus, dass der überwiegende Anteil der Zeit von der Featureextraktion beansprucht wird, genauer gesagt für die Stringkonkatenationen. Die Featureextraktion ist allerdings mit einigen Parsingstrategien nicht in jedem Zustand notwendig; daher ist es nicht ausreichend die Anzahl der Zustände zu vergleichen, die verschiedene Strategien benötigen. Stattdessen muss die Anzahl der eigentlich notwendigen Featureextraktionen verglichen werden. Diese Zahl ist für die Algorithmen von Nivre als auch für die Parsingstrategie von Covington ähnlich, auch wenn sie unterschiedliche Komplexitäten haben. Außerdem können Techniken wie Memoisierung, die die Wiederverwendung von Features ermöglichen, so dass teure Stringoperationen nicht unnötig wiederholt werden müssen, leichter für Covington's Strategie als für Nivre's lineare Algorithmen angewendet werden, was dazu führt, dass Covington's Algorithmus trotz schlechterer worst-case Komplexität eine bessere Leistung liefert.

Beim Einsatz von Baumbanken bin ich zu sehr interessanten Erkenntnissen gelangt, die sich auch in dieser Arbeit wiederfinden. Baumbanken sollten eigentlich von sehr hoher Qualität sein, weil sie ausschlaggebend für den Erfolg aller datenbasierten Ansätze sind, deren maschinelle Lernverfahren auf ihnen aufbauen. Ich konnte zeigen, dass dies häufig nicht der Fall ist und Baumbanken eine bedeutende Menge an fehlerhaften Annotationen enthalten. Das hat zur Folge, dass kompliziertere Verfahren teilweise nur deshalb besser bewertet werden, weil sie diese Inkonsistenzen besser wiederholen können. Die erzielten Ergebnisse sind dann trotz der besseren Wertung objektiv nicht von viel höherer Qualität. Ein von mir entwickelter Ansatz kann nicht nur die Inkonsistenzen erkennen, sondern diese auch korrigieren.

Für einige Sprachen sind sehr große Baumbanken vorhanden, allerdings zeigen Lernkurvenexperimente, dass ein annähernd gleiches Ergebnis schon mit weniger Daten erreicht werden kann. Für andere Sprachen sind Baumbanken zu klein und mit mehr annotierten Daten könnten viel bessere Ergebnisse erreicht werden. Da die Annotation von Korpora kostspielig ist, ist es wichtig, den Annotationsprozess so gut wie möglich zu kontrollieren. Daher stellt sich

die Frage, welche Art von Daten nützlich ist und zuerst annotiert werden sollte, da sie sowohl mehr Nutzen für die Performanz als auch den Gesamtaufwand für die Annotation reduziert. Diese Frage habe ich in meiner Arbeit erfolgreich untersucht.

Ich habe mehrmals die Gelegenheit ergriffen um mit meinen Systemen an Shared-Tasks für Erkennung von Textual Entailment teilzunehmen. Dabei war mein Ziel, meinen Parser auf konkrete Aufgaben anzuwenden. Die erste Task, PETE, war speziell darauf ausgerichtet, Parser zu evaluieren und stellte interessante Daten zur Verfügung, die die Erkennung einer begrenzten Anzahl von Dependenzen erforderten. Die beiden Tasks RTE-6 und RTE-7 waren nicht für Parserevaluierung konzipiert, ich konnte die mitgelieferten Datensätze trotzdem nutzen um den Parser auf realen Daten zu testen. Diese Aufgaben ermöglichten es mir besser zu verstehen wie die Integration von Parsern in Anwendungen funktioniert. Darüberhinaus zeigen diese Experimente, dass sich die Performanz von Parsern auf den üblichen Entwicklungsdaten der Baumbanken von der auf realen Daten deutlich unterscheidet. Ich habe bereits darauf hingewiesen, dass ein großer Mangel der aktuell vorherrschenden Evaluierung darin liegt, dass die Performanz der Parser als Mittelwert über alle Dependenztypen berechnet wird. Einige aktuellere Publikationen gehen auf diese Tatsache ein, in dem sie die Evaluierung auf eine für die anvisierte Aufgabe besonders wichtige Teilmenge von Dependenztypen einschränken. Ich habe eine ähnliche Analyse für die Erkennung von Textual Entailment durchgeführt und präsentiere im folgenden meine Ergebnisse.

Viele Teile dieser Arbeit wurden auf verschiedenen Konferenzen veröffentlicht. Auf diese Weise sind neben einem gut funktionierenden Dependenzparser sieben Papiere entstanden, die auf internationalen Konferenzen begutachtet und akzeptiert worden sind.

1 Introduction

Parsing is the process of mapping sentences to their syntactic representations. These representations can be used by computers for performing many important natural language processing tasks, such as question answering, information extraction, coreference resolution, textual entailment, machine translation and many others. Dependency parsing is a parsing technique, which is based on dependency grammars (DGs)[38] - a theory of sentence structure. During the last decade the area of this grammar and methods based on it have been proven very useful, because of the ability to reliably capture very valuable information within a sentence in a very convenient easy-to-use form. Moreover, compared to other theories of syntax, dependency representations have a lot of appealing properties, which I am going to present in this work.

A lot of different groups of researchers all over the world have developed a number of good parsers over the time, e.g. MaltParser, Minipar, MST Parser, Stanford Parser, Ensemble, Mate-tools or Clear Parser. Parsers are very complex systems, with many different properties. Each of these parsers is therefore somehow special and has its own unique characteristics. In this thesis I am presenting yet another new dependency parser - MDParser (Multilingual Dependency Parser), whose ultimate goal is to be particularly performance-oriented. In the thesis I will discuss that most current systems focus on accuracy, neglecting efficiency. In this work I will focus on efficiency, however, not neglecting accuracy. I will refer to this combined focus as performance-orientedness. The focus on efficiency is very important, because a lot of new applications work with the data from the web or have to be able to do the processing online, rather than preprocessing some data offline once, so that it can be used by an application later. In order to guarantee multilinguality I have restricted myself to data-driven parsing only without any language specific modifications to the system and thus I will not elaborate on rule-based or other hand-crafted dependency parsing approaches.

I will present an overview of important features for parsers and discuss their properties and effects on the system and its results. In particular I will describe the choice of properties for MDParser and elaborate on the differences and provide motivation for my decisions. I will try to present a decent summary of the progress in the field of dependency parsing, as well as point out and discuss the trends. Finally, a big portion of my thesis will be dedicated to dependency parser evaluation, a very important topic, in order to truly understand the

quality of different systems.

A lot of the work presented in this thesis has been published on various conferences. Thus besides a well-working dependency parser, there are also seven papers, which have been peer-reviewed and accepted on different international conferences, which arise from this work.

1.1 State of the Art Dependency Parsing

Dependency parsing enjoys great popularity in the research community. The huge interest on the topic is very well noticeable, when looking at some shared tasks (among most renowned are CoNLL-2006, CoNLL-2007 and CoNLL 2008 tasks[15]), countless publications at all major conferences (ACL, COLING) and numerous PhD works. When so many researchers work on the same topic it becomes difficult to generalise about different tendencies in the field, since many groups develop their own views on different aspects of the area. Still, I will try to summarise the current state of research, as well as the trends, of dependency parsing and critically discuss them.

Parsing is seen as a goal in itself This basically means that the task is seen independently of what is happening before or after it. On the one hand parsing usually requires preprocessing steps, such as sentence boundary detection, tokenisation and POS-tagging. Their performance can have a crucial influence on the quality of parser results. On the other hand parser results are usually used in follow up applications, such as coreference resolution modules, for correct translations or for information extraction. Not all information is relevant and not all relevant information is of equal importance for these tasks.

However, the state of the art seems to ignore these aspects. The necessary preprocessing is usually taken for granted by working with treebanks, which have all the information available. The usability of the parser results is usually not considered neither, which is best illustrated by the predominant evaluation metrics, such as attachment scores, which treat all dependency relations equally, whereas for applications certain aspects of the syntactic structure are in general more important than the others, e.g. semantically meaningful subjects or objects vs. purely syntactic ones, like determiners or punctuation.

Evaluation metrics are very dull The most spread evaluation metrics are the labeled and unlabeled attachment scores (LAS and UAS), which measure

the proportion of correctly recognised dependency relations with both correct head and label in case of LAS or only correct head in case of UAS.

These scores do not suffice for a comprehensive evaluation of a parser. A gold standard dependency treebank is required and there are only few of them for most languages. This fact has several consequences:

- the parsers are tuned to a specific treebank, which might impair their performance for other domains.
- the scores do not differentiate between various dependency relations of different importance.
- meanwhile the parsers achieve very high scores for these specific treebanks and these scores seem to stagnate lately. This fact suggests that parsers perform very well and that no improvements are being achieved. However, some recent work (e.g. SemEval-2 shared task PETE[87]), as well as this thesis, clearly show that parsing technology is far from being that reliable when applied to data different from these treebanks and that there is still a lot of room for improvement.
- treebanks inevitably contain errors[31]. Whereas overall these resources are of high quality and the errors probably do not have a significant effect on the performance, these wrongly annotated tokens still falsify the evaluation result. Therefore a natural evaluation[87], rather than an evaluation based on artificially tagged data, provides a more realistic feeling for usefulness of parser results, than attachment scores do.

Parser performance implies only the quality of results When the performance of a parser is being described, usually only the quality of the results is provided, e.g. using the above mentioned attachment scores. The parsing times and the training times for parser models are considered and provided much more rarely (cf. CoNLL challenges where the parsing speed is not an issue). Moreover, because of the stagnating accuracy scores, there are some recent trends to combine different parsers[63][1] in order to achieve better results, i.e. higher attachment scores. Whereas some of these combinations are based on parallel architectures, e.g. simultaneous parses with different systems with a subsequent voting (cf. Ensemble[77]), the "real" improvement (current

[1] This is an invited talk, last retrieved March 10,2011. http://stp.lingfil.uu.se/~nivre/docs/Current.pdf

improvement is around 1-2% in terms of attachment scores) is expected only when systems are run one after the other, when one system can use features based on the output of the previous one[66]. It is obvious that such a system will have much lower running times. It seems that no trade-off between speed and accuracy is being made and the quality of the results is the ultimate goal. Additional properties like the amount of necessary data for training or memory requirements are not an issue in current works at all.

Extrinsic evaluation possibilities are limited Due to the above mentioned disadvantages of current evaluation metrics there is a growing number of alternative evaluation attempts. Extrinsic evaluation is one such possibility. The idea is not to measure the performance of a parser directly, but rather embed it in some NLP task, for which dependency parsing is beneficial, and then measure the improvement. The bigger the improvement, the better the parser. The already mentioned PETE[87] shared task for recognising textual entailment or parser evaluation for information extraction[58][17] are examples of such evaluation. The problem with this evaluation is that such embedding is not easy to do and requires a high level of expertise in the task, as well as considerable implementation effort. Furthermore, the embedding introduces an additional level of complexity to the evaluation. It becomes not clear whether the impact on the final result is due to the quality of the parser or due to the quality of the embedding. The embedding might also favour one parser and its result more than some other, which does not necessarily mean that this parser is better, since with a different embedding the results could look different.

Evaluation of individual dependencies is a compromise Both intrinsic evaluation with attachment scores and extrinsic evaluation with difficult embeddings have their disadvantages. Some of the recent alternatives therefore propose a compromise between both. The idea is to use attachment scores, because of their robustness, but to apply them only to a very limited set of dependencies, that one is particularly interested in, e.g. because they are relevant for one's application. Examples for this work are the evaluation of non-local[8] and unbounded[69] dependencies. This method is a real alternative, since it copes with drawbacks of both intrinsic and extrinsic evaluation. I will perform a similar evaluation for the task of recognising textual entailment for the RTE-7[10] shared task data.

1.2 Contribution of this Thesis

In my description of the state of the art dependency parsing I have worked out some points of criticism:

- Parsing is treated as an independent task and not together with applications, for which it is actually being done.
- The quality of results is not measured properly. The evaluation is being done on the same data for years and the extrinsic evaluation is difficult and rare.
- Parsing may not be expensive, if it is to be used in follow-up applications. However, the costs and requirements of parsing are very often neglected by the community.

In this thesis I provide a comprehensive overview of different parsers and their properties. I will then introduce a particularly performance-oriented parser MDParser, which I have implemented in the course of this work. The performance-orientedness means that it is both fast and sufficiently accurate for applications, not only accuracy-oriented as most of the systems I am going to introduce. Morever it is able to process data in its plain form, without having to transform it to a specific format and is in general easy to use. Finally, I will elaborate on the topic of parser evaluation, which is definitely undervalued in the recent literature. The most important topics of parser evaluation include approaches on:

- How to avoid tuning to treebank errors.
- How to evaluate a parser only on relevant relations.
- Whether linear or non-linear kernel-based parsers are better qualified for applications.

The goal of this thesis is to enable researchers to reliably select the parser which suits their goals best, when developing some NLP application. For practical applications it should be a considerable improvement to the currently common practice of looking at the CoNLL rankings or other WSJ evaluations and choosing the parser with the best score.

1.2.1 Thesis Achievements

The most important successful achievements of this thesis are:

Efficient feature extraction I have used Java profiling technology in order to analyse how much execution time is spent in each stage of processing. The main finding was that the overwhelming amount of spent time accounts for feature extraction. Even though this fact has already been reported by others in the literature for graph-based parsers[12], for transition-based parsers the common understanding is that the efficiency is mainly dependent on the complexity of the parsing strategy[61], i.e. the number of states the parser requires to parse a sentence in the worst case. However, my experiments have shown, that this is not true. First, the worst case does not occur in practice, since human language does not favour constructions which are hard to parse[25]. Therefore the complexity is not very significant and it makes much more sense just to take some empiric metrics, like the average number of states per sentence. Second, even a larger number of states does not matter, if one spends less time per state than with a strategy with a small number of states, but high costs per state. Since the costs are mainly dependent on the time spent on feature extraction the most efficient strategy is thus a strategy, which allows the most efficient feature extraction. In this thesis I will demonstrate that the Covington's strategy with quadratic complexity, which I found most suitable for efficient processing, greatly outperforms the usually preferred Nivre's algorithms with linear complexity, which are not suitable for efficient feature extraction.

Error identification and correction Every treebank contains annotation errors, which are harmful for data-driven parsers, which learn to replicate them. Despite the fact that the quality of annotated resources is of essential importance for the quality of resources derived from them, there is not much literature available on this topic. A notable exception is the group around Detmar Meurers and some publications about their approach for finding errors called *variation detection*[30]. I propose a different method, which is complementary to variation detection and is able to find additional errors. Moreover, it is not only able to recognise inconsistencies, but also proposes a correction. The proposed method was published in the proceedings of ACL 2011[82].

Embedding and evaluation of dependency parsing for the task of recognising textual entailment Extrinsic evaluation is an alternative to the usual tests on the held-out data of a treebank using attachment scores. In order to gain experience in that field I have participated into two RTE tasks: PETE and RTE-6[9]. PETE was a shared task designed specifically for parser

evaluation and contained a small amount of data with different types of dependencies, which were used to evaluate parsers. RTE-6 did not have the purpose to evaluate parsers, but it contained a large amount of real-world data with all kinds of dependencies, which I could use for parser evaluation. The result of this work is that I was able to build a sensible embedding of parser results into an RTE system and evaluate numerous parsers with this system.

Experiments with sentence usefulness Accurate data-driven parsing requires large treebanks with high-quality annotation. Such treebanks require a lot of effort, since double or even triple annotation of large amounts of data is necessary. Therefore resources of sufficient quality are available only for few languages. Since multilinguality is very important for NLP tasks, I was interested in investigating how the problem of insufficient annotated data can be solved. One possibility is to speed up the annotation process by annotating those sentences first, which are most useful for the task and do not annotate those which are of no use. I will describe an approach how such sentences can be found automatically.

Investigation of differences between linear and non-linear kernel-based learners In the recent years accurate parsers have been slow and fast parsers have been relatively inaccurate. Currently, fast parsers are based on linear learning methods and are just one percent behind accuracy-oriented slow parsers, which are based on sophisticated non-linear kernel-based learners. In this thesis I will show the investigation whether it is worth suffering the disadvantages of hundred-fold lower training and parsing times to gain this one percent of accuracy. This analysis consists out of investigation of results obtained with both approaches. Therefore I analyse the tokens which are correctly recognised by one, but not the other approach. I investigate whether these tokens are just an ordinary random extract of the data or are somehow special, whether non-linear classification performs better simply due to overfitting the data and whether a more complex approach helps to recognise more difficult dependencies.

Evaluation of relevant dependencies Finally, I have tried to combine intrinsic and extrinsic evaluation to benefit from their advantages and avoid their disadvantages. For this goal I have built a framework, that I am going to describe in details in the main part of the thesis, and applied it to a small portion of RTE-7 data. Afterwards I performed parser evaluation with several parsers.

This way I have tried to benefit from the useful properties of attachment scores and to avoid the drawbacks of extrinsic evaluation that I have encountered in the PETE and RTE-6 shared tasks.

1.2.2 Trial and Error

In the course of three years of work on this thesis I have investigated numerous approaches to improve my parser. The overwhelming majority of these attempts were failures or became obsolete in the course of the time. Nevertheless, I want to elaborate at least on some of them in this thesis. Of course they can not be seen as a real contribution since they brought no improvement and are not part of the current system. Nevertheless, without having analysed why they have failed and lessons learned out of them, I would have never been able to arrive at the final solution for MDParser.

The most important pieces of such work included:

Experiments with different machine learning packages In the course of the years I have tried out four different machine learning algorithms: OpenNLP MaxEnt[39], Mallet MaxEnt[52], L1 regularised logistic regression[85] (LibLinear package)[46] and finally multiclass support vector machines by Singer and Crammer[42] (LibLinear package). The final approach delivered the best accuracy and efficiency, but other algorithms and packages also played an important role during the development.

Experiments with different parsing strategies There are numerous transition-based parsing algorithms. I have experimented with many of them, including the most important ones like Nivre's arc-standard and arc-eager parsing strategies[62], but finally decided to stay with Covington's fundamental parsing strategy[25]. I then tried to improve Covington's algorithm with many ideas, most of which failed. Among such attempts was a divide and conquer modification to the algorithm, which later became obsolete because of better classifiers or additional transition types introduced to the strategy.

Experiments with feature usefulness Many older state of the art parsers like MST Parser[53] or MaltParser[65] using LibSVM[20] took many hours or even days to train a model for a larger treebank as the one for English. Feature engineering was thus an extremely tedious time-consuming occupation. However, it was indispensable, since only with a good model good results could

be achieved. Most approaches of those days included training with less time-consuming machine learning strategies, e.g. TiMBL[26] in case of MaltParser, and then retraining with the best found features with a more sophisticated learner. Whereas this method usually worked quite well, a good feature for one learning algorithm does not necessarily have to be good for a different approach and thus the models were still not optimal. Additionally, the less complex algorithms still required some time to train a model. I have concentrated on a different method, which allowed to estimate feature usefulness without training a model and spent a lot of time on its development. In the end it turned out that newer learning strategies, which implicitly contain feature selection, outperformed my approach.

Experiments with splitting Instead of training one classifier for the whole training data, it is possible to split the training set into parts and train many classifiers. The requirements are that you know when to apply which classifier and that there is enough data, so that smaller pieces of data are still big enough in order to allow accurate training. Otherwise one can arbitrary divide data in as many pieces as desired. Training on smaller training sets is done much faster and sometimes even provides an accuracy boost because data separability is increased. In MaltParser splitting is done in a such way that the training data is split according to the value of some feature, e.g. POS tag. This way one gets one model for verbs, one model for nouns, one model for adjectives etc. I have initially tried a different approach. The idea was to split the data purely by size, train the models, estimate the most useful features with the above-mentioned technique and then train a model for the whole data without splitting, but only with the estimated useful features. The reason for why I did not want to use splitting for the final model was that for many languages the treebank size is not sufficient even if the whole data is used and splitting can only deteriorate the result. However, since feature usefulness estimation became obsolete, this way of splitting did not make sense any longer neither. Thus the final version of MDParser contains splitting as it is realised in MaltParser.

Experiments with confidence values Parser results are often inaccurate for out-of-domain data. In some cases it is better not to use parser results at all, since erroneous parser decisions can mislead the system in which the parser is embedded. Therefore it is helpful to know how confident the parser is about its decisions, so that only reliable results are used. I have tried to estimate such

confidence values with help of probabilities that I could compute with parser's model. Unfortunately, often even a 100% probability for a certain dependency according to parser's model leads to erroneous decisions and even though it was much less often the case than when the probability was lower, e.g. 80%, the results were still not satisfactory.

I will elaborate on these experiments in more details in the main part of the thesis.

1.3 Outline

The thesis consists out of two parts.

In the first part I introduce the fields of dependency grammar and parsing. I discuss different parser properties and present the most prominent dependency parsers in Section 2. I then introduce MDParser, the dependency parser I have developed in the course of working on this thesis. I describe its architecture and properties, as well as point out the differences to MaltParser, on which it is based in Section 3. I conclude the first part by presenting all my experiments in the field of dependency parsing, both successful and unsuccessful ones and discuss their contributions in Section 4.

In the second part I present the evaluation methodologies available for dependency parsing, discuss their advantages and disadvantages, as well as the necessary modifications to account for their shortcomings. The intrinsic evaluation methodology is covered in Section 5 and the extrinsic approach is described in Section 6. Finally I evaluate MDParser, as well as some other state of the art systems in Section 7.

Section 8 concludes the thesis, summarises its achievements and addresses the future directions for development, which come short in this thesis.

Part I

Dependency Parsing

2 Overview

Dependency grammar and dependency parsing have a long tradition. Most of the ancient linguists assumed some kind of dependency representation for their languages, e.g. ancient Greek, Sanskrit, Latin, Arabic[60]. In medieval Europe and 1900s there were also a lot of proponents of dependency grammar. Tesniere introduced the notions of head and dependent in 1959[78]. In 1988 Melcuk adopted dependency grammar to Slavic languages[54] and the formalism more or less corresponds to what is now used by most dependency parsers, the main difference being that dependency parsers only produce one layer of syntactic dependencies, whereas Melcuk also distinguished among other layers like morphological dependencies. I will describe this formalism in this section and define exactly what properties I assume.

In the last decades dozens of dependency parsers were developed. Dependency parsers are complex systems with many different properties, so I will try to provide an overview of these, discuss their importance and name corresponding parsers or literature. This is necessary in order to introduce my own parser later on. This section will thus provide all necessary terminology and definitions, which will be used to describe the core of the dissertation in the following parts and is thus essential for their understanding.

2.1 Dependency Grammar

In this subsection I am going to describe how a dependency representation looks like and what are its formal properties.

Dependency grammar works only with surface forms of the words in the sentence, without assuming any phrasal units. This is a very special property of dependency grammar, since there is no way to treat particular groupings of words (such as chunks or constituents) in a specific manner. Even though these surface forms do not necessarily have to be identical with the word forms we observe in a sentence, I will nonetheless assume it in this thesis and will refer to these units of the syntactic analysis simply as words. The only exception will be the special root node (0). Words are interconnected by special meaningful

relations called dependencies and form a structure called dependency structure. Here is an example of dependency structure:

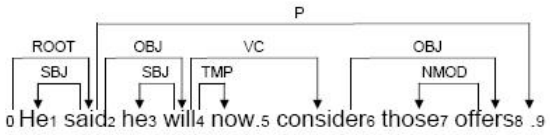

Figure 1: Dependency structure

Dependency and dependency structure are central notions for the theory, therefore it is necessary to list the properties I assume for them in this work.

2.1.1 Dependency Relations

Dependency relations are:

- **binary** (only defined for pairs of words)
- **directed / antisymmetric** ($A \rightarrow B \neq B \rightarrow A$; the direction between the words is important, furthermore I assume that for any pair of words a relation only in one direction is allowed at the same time)
- **anti-reflexive** (if $A \rightarrow B$, then $A \neq B$; no word can govern itself)
- **anti-transitive** (if $A \rightarrow B$ and $B \rightarrow C$, then $A \nrightarrow C$; dependency relations are only direct)
- **labeled** ($A \underset{\rightarrow}{l} B$; the relation between two words is typed with the label l)

Since dependencies are binary, I will often use the notation used for pairs - (A, B), which is an abbreviation for $A \rightarrow B$. The term I will use for A is *head* or *parent* and B will be called *dependent* or *child*. Even though dependencies are not transitive, it is still sometimes useful to know whether some element is dominated by another indirectly, i.e. in several steps. Therefore I will use the notation $A \underset{\rightarrow}{*} B$ and will call B *subordinate* of A.

2.1.2 Dependency Structures

Here are some properties of dependency structures:

- **connectedness** (all words should be dominated by at least one other word)
- **single-headiness** (all words, except the special 0 word have to have exactly one head)
- **rooted** (each sentence has exactly one token whose head is 0)
- **acyclicity** (there should be no way that $A \xrightarrow{*} A$, in other words there should be no cycle)
- **projectivity** (if $A \rightarrow B$, then all words between A and B have to be subordinate of A; in terms of the dependency graph as seen in Figure 1, it means that there are no crossing branches)

First four properties guarantee that dependency structures are actually dependency trees, a data structure that allows very efficient algorithms. The fifth property is very controversial, since non-projective structures occur in many languages. However, for the sake of efficiency, most parsers assume projectivity and even for an algorithm, that is designed in such a way that non-projective structures can be captured, projectivity is often optional, because it affects its complexity. The other properties, both of dependency relations and structures can as well differ for different dependency grammar variations, e.g. relations are not antisymmetric in every formalism or words can have more than one head, however I will not elaborate on the motivation for it and stay with the version of dependency grammar described in this section, because it is most widely-used and most efficient from the computational point of view.

2.1.3 Strengths and Weaknesses

It is crucial to differentiate between the advantages and disadvantages of a theory and of a certain system based on this theory. Obviously a good system will always incorporate the benefits, that a theory can provide, and at the same time try to cope with its weaknesses, however, it does not necessarily have to be the case. For instance, one of the main properties of dependency grammars, which explains its popularity for some languages (e.g. Slavic languages) is its ability to deal with the free word order in a very easy fashion. However, this property

is not exploited in most of the currently available systems for computational reasons. On the other hand, some of the points, that are traditionally considered to be weaknesses of the theory, do not necessarily have to be weaknesses of a system. For instance coordination structures, that are considered to be problematic constructions for dependency grammar, since the theory can not provide a unique solution how to treat them. However, a system will probably cope with any of the possible ways of representing them equally well, as long as the representation remains consistent throughout the whole data.

In this section I will try to answer the question, why dependency grammar has become so popular recently. To my mind it is best done by analysing its advantages and disadvantages. Therefore I want to enlarge on different important phenomena and how good or bad our theory can handle them.

Strengths Let us start with listing the advantages of the theory:

- Dependency analysis does not require any phrasal (non-terminal) nodes. It only works with existing surface forms (e.g. word forms) and tries to find relations between them. This makes the parsing task more straightforward and the algorithms clearer and less complex.

- Dependency parsing provides an analysis with very useful relations. First of all the relations are explicit - it is always clear which element is the head and which element is the dependent. Furthermore, this relation is also typed, and thus it is not necessary to guess what kind of dependency is between them, which is a source for numerous kinds of ambiguity otherwise. Additionally, these relations are very close to semantics (argument structure vs semantic roles), which is almost always the next step after syntactic analysis in natural language processing.

- Dependency representations provide structural order, not linear order. This way discontinuous phrases, and other phenomena typical for rather word free languages, can be treated elegantly. This property is almost unique among syntactic theories, which heavily rely on linear order for their analyses.

- Dependency parsing is very suitable for incremental processing, since it is possible to process words one by one as soon as they are encountered. It is not necessary for the analysis to wait until a certain fragment (e.g. a phrase) is finished.

- Dependency structures on many different description levels of the language can be constructed. For some languages it is convenient to have a layer of morphological dependency relations between words (e.g. *I write* vs *he writes* - verb governs the subject syntactically, but is dependent on it morphologically), sometimes it is useful to have a layer of semantic relations (e.g. *it depends on the weather* - syntactically the object of *depends* is *on*, and *weather* is then the object of the preposition, semantically, however, there is a direct relation between *depends* and *weather*) and yet in other cases it is sensible to have a discourse-related representation (e.g. *he bought a book and read it* - there is a relation between *book* and *it*).

Weaknesses Now let us take a look at some problematic cases for dependency grammar:

- Dependency grammar does not provide a unique solution for coordination structures. Both parts of a coordination are obviously equal, however, the theory only offers head-dependent relations. Especially if one assumes the single-head constraint, then it is unclear how to capture this kind of structure in a reasonable manner.

- Dependency grammar offers relations only between individual elements, it does not have a notion of groupings or constituents. However it is obvious that some phenomena occur not on the word level, but on the phrase or sentence level. The probably most frequent phenomenon is modification, which can occur for single words as well as for large phrases. In those cases it is impossible to tell what kind of dependency relation it is just by looking at it (e.g. *I have lived there last year* - *last year* can either only modify *lived* or the whole sentence).

- Dependency grammar also does not provide a uniform solution for auxiliary verbs. It is unclear whether the auxiliary verb should govern the main verb or the other way around. It is also sometimes unclear how to deal with numerous auxiliaries occurring at the same time.

A rough analysis of the strengths and weaknesses of this theory explains why the theory is so popular and why I have decided to work with it in my thesis. Dependency grammar allows us efficient parsing techniques and produces a meaningful output, which can be used for further tasks. The problem that it is linguistically not always ideal is unpleasant, but an acceptable compromise.

2.2 Dependency Parsing

Parsing is the process of assigning a syntactic analysis to a sentence. In dependency parsing the analysis is based on the dependency grammar. There are numerous ways of getting dependency representations. The most important approaches are rule-based dependency parsing, data-driven dependency parsing and the derivation of dependencies from a different syntactic theory.

In the rule-based approach a formal grammar describes the language which can be parsed and how the structure of its sentences looks like. A grammar parsing algorithm then is able to determine whether a given sentence belongs to the language or not. In positive case the derivation tree is the dependency analysis of the sentence. The main challenge of this approach is to provide a grammar of sufficient coverage, since it is difficult to provide enough rules in order to accept all possible sentences and at the same time avoid introducing ungrammatical structures. Another challenge with this approach is disambiguation, since sometimes the same sentence can have several possible derivations and a mechanism determining the most probable one is necessary.

In the data-driven approach the mapping from strings to structures is induced from the data. A data-driven parsing algorithm then is able to construct different analyses, score them and select the most probable, according to what is most reasonable given the data it has seen before. Thus data-driven parsing can be split in two phases: the *training phase*, when the mapping is learned, and the *parsing phase*, when the mapping is applied to get the most probable structure. Rule-based parsers have good accuracies, but only for the sentences which are covered by the grammar, whereas the data-driven parsers accept any string given and try to make best out of it. The drawback is that even ungrammatical sentences get an analysis, but coverage is no longer a problem. For good accuracies, however, a lot of data is necessary, which is the main challenge of this approach.

Both approaches are inherently doing the same thing: the parsers are given rules, apply them and derive a structure. The difference is how one gets the rules: in the data-driven approach they are induced from the data and in the grammar-based approach the rules should deduce the data. The difference is thus as with inductive and deductive reasoning. One can simplify it by saying that rule-based systems initially have perfect accuracy and grow in coverage during their development and data-driven systems initially have perfect coverage and then grow in accuracy[61].

From the technical point of view the data-driven approach is more appealing, since the main labour with this approach is the preparation of the appropriate data, which usually has already been done by someone else. In the rule-based approach the developer of the parser has to develop the set of rules, as well as their weights and priorities manually, whereas with data-driven approaches it is automatised by machine learning techniques. Especially, if one desires a multilingual parser it is only possible with the data-driven approach, since one and the same parser can work for various languages, provided the data. On the contrary, a rule-based system needs language-specific grammars for each language, which is an unmanageable problem.

The third possibility of deriving dependency representations does not strictly belong to the field of dependency *parsing*, because the underlying parsing process is based on a different syntactic theory. However, it is still relevant, because it allows to derive the same representation. For example one can derive dependencies from phrase structures by applying head rules[19] or one can derive them even easier from other deeper formalisms like LFG[40] or HPSG[72], since their analyses contain even more linguistic information to construct dependency trees out of them[88].

Since we are interested in the efficiency-oriented parsing for applications, and applications often require multilinguality, I will focus on the data-driven parsing in this thesis. Therefore I will define the training and parsing phases for this approach in more details.

Depending on whether the data one wants to learn from is annotated or not, the learning can be supervised or not supervised, respectively. I will restrict myself to supervised parsing only because supervised dependency parsing is a mature technology, whereas unsupervised dependency parsing is still a relatively unexplored area. Therefore it would be to difficult to perform a reasonable evaluation and meaningful comparison of my work with other developments.

There are two possibilities how one can proceed in order to produce dependency representations given the appropriate data: graph-based and transition approaches.

Algorithms belonging to the category of transition-based parsing strategies deliver the parse of a sentence after performing a sequence of actions one after another. The final result should be the set of all dependency relations, required to construct the correct dependency tree. The module responsible for choosing the best operation in every step is called *oracle*. At first, during the training phase, the perfect oracle is simulated by using the training data, during which

the system is driven by the already given gold standard result. During this phase the system learns which operation is likely to be chosen in which situation. These situations depend on the current state of the system and its auxiliary data structures, and are called *configurations*. The result is a model, which is then used to make predictions about the most likely action, when the gold standard result is unavailable, which happens in the parsing phase. An example for a transition-based parser is again MaltParser.

In graph-based parsing the algorithms construct dependency graphs for a given sentence, typically by initially assuming edges between all words and then eliminating all wrong ones until the graph does not become a valid dependency tree with the maximum global score according to the model. The model is approximated in the training phase by learning the edge weights from the data. This approach is very different from the transition-based parsing, since it finds the solution in several global steps, which involve the information about the entire sentence, whereas transition-based algorithms use only local information and require a long sequence of parser decisions in order to arrive at the final result. MST Parser is an example for a parser of this class.

2.3 Parser Properties

In this section I am going to list all relevant properties which have to be specified when developing a dependency parser. The variety of these implementational decisions leads to the sizable number of different systems in the field. For each property I will provide some sample values and give an example system or cite the corresponding literature.

Grammar This determines the process how dependencies are derived. In a rule-based system a dependency grammar is responsible for deducing the structure. A good example is the MiniPar system for English[49]. In a data-driven system the structure is induced from the data. There are numerous systems of this kind, most prominent being MaltParser[65] and MST Parser[53]. The dependencies can also be derived from a different formalism, e.g. as in Stanford Parser[44], where dependencies are extracted out of phrase structures.

Machine Learning This determines what kind of machine learning approach is used in the system. Only few systems are purely rule-based, most of them, even if they have an underlying hand-written grammar, still rely on statistics

learned from the data, e.g. for disambiguation. An example for such system is the parser for German Pro3Gres[74]. As far as the data-driven parsers are concerned, it is self-evident that they are highly dependent on the machine learning method. Here, there are again numerous possibilities to choose from. The field of machine learning is so vast, that I will not be able to give a comprehensive overview on all available techniques, but I will point out the main properties of a machine learning approach for dependency parsing.

multi-class vs binary: Dependency parsing is highly dependent on multi-class classification. For instance there are usually dozens of different dependency types and so the system must be able to select the right one. Most classifiers are binary and can not solve this task directly. However, with one-versus-one or one-versus-all strategies one can indirectly construct a multi-class classifier. Classification for the one-versus-all case is done by a winner-takes-all strategy, in which the classifier with the highest output function assigns the class. The classification of one-versus-one case is done by a max-wins voting strategy, in which every classifier assigns the instance to one of the two classes, then the vote for the assigned class is increased by one vote, and finally the class with most votes determines the instance classification. However, there are also classifiers, which are able to simultaneously differentiate between many classes, so that no workaround is necessary.

probabilistic vs non-probabilistic: A probabilistic classifier computes a probability distribution for a given feature vector. A probabilistic approach might be beneficial, if one wants to fine-tune a system, e.g. by preferring some important class, even if it does not get the highest probability, which is often the case when the training data is imbalanced. It can also be used as a confidence value for classifier's certainty about decisions or in order to look up the competing class, i.e. the class with the second-best probability, which might also be helpful in certain situations. With a non-probabilistic classifier one does not have these possibilities, since it merely returns the predicted class.

linear vs kernel-based: Linear classifier identifies the class by a linear combination of all features in a feature vector, which does not require a lot of computation. One can visualise its operation as dividing one class from another by drawing a line between them. Of course this assumes that such line can be drawn, i.e. that the data is linearly separable. Usually this is not the case. A

kernel-based classifier makes use of the method called kernel trick [5]. According to this method a linear classifier solves a non-linear problem by mapping the original observations into a higher-dimensional space, where the linear classifier is subsequently used. The mapping is achieved by applying a kernel function to the feature space. Wherever a dot product is used, it is replaced with the kernel function. This is much more expensive, but guarantees better separability.

Determinism The understanding of the notion in computer science is that a deterministic algorithm always produces the same result, i.e. there is no random component in it. In the field of parsing, however, the notion usually means something different, namely that the algorithm does not back up[4]. Especially due to ambiguities in the language it is difficult to predict the correct structure right from the start, without having to adjust the result in the course of processing. Therefore many parsers allow non-deterministic behaviour, e.g. by backtracking when there are new indicators, that something done in the past has gone wrong, or by beam search, i.e. keeping many possible solutions in memory simultaneously as long as they are plausible. Yet another variation of non-determinism is reranking or repair, i.e. the result is changed not during the processing, but afterwards. In case of reranking an n-best-list of possible parses is required and in case of repair the result is evaluated by a different model and corrected if necessary. This is different from the technique used by many deterministic parsers called postprocessing, when situations with uncertain decisions are postponed for a later point, but if a decision is done it is never revised. An example for parsing with repair is the work by Hall and Novak[36]. A beam-search-based dependency parser is the system described by Zhang and Clark[89]. Among deterministic parsers is the already mentioned MaltParser and the work by Ytrestøl describes a method for non-deterministic dependency parsing with backtracking[84].

Incrementality Real-time applications often require online processing of the input received so far. Methods, which are able to produce partial structures for an incomplete input and require only one left-to-right (or right-to-left) pass in order to construct the final analysis are called incremental. On the contrary to that, parsers which require the input before the processing are non-incremental. Incremental parsers are rare because, especially in the deterministic scenario, no reliable structure can be produced without knowing what input is still to come. However, in many cases there is no need to wait for the entire sentence

to be read in. The knowledge about few next words, so called lookahead, is already very helpful and is sufficient for high quality results. The exact size of the lookahead varies for different languages, but experiments show that 3-4 words are usually enough. An example of a fully non-incremental parser is MST Parser, since it requires the whole sentence to be read in order to compute the maximum spinning tree. All state of the art transition-based parsers use a lookahead.

Language Parsers can be language-specific, language-independent or language-tuned. Language-specific parsers work only for one language. MiniPar for instance, which is a grammar-based English parser will not work for any other language. Language-independent parsers work for any language and achieve optimal results for that language without fine-tuning. Language-tuned parsers work for any language with some default settings, but the results can be significantly improved if language-specific adaptations are made. Such fine-tunings include learner's parameter optimisation, language-specific feature models or even different parser strategies. Most data-driven parsers are of this kind. Unfortunately there are still no truly language-independent parsers, which automatically adopt themselves to the given language and reach an optimal accuracy.

Parsing Strategy Parsing consists out of two steps. In the first step the rules/models are learned and in the second they are applied in order to get the analysis. There are two fundamental approaches how the analyses are derived: graph-based and transition-based. Both approaches then have numerous different algorithms.

There are several prominent transition-based algorithms, which are very well summarised in a journal article by Nivre[62]. They can be broadly divided into stack-based algorithms, like Nivre's arc-eager and arc-standard algorithms on the one hand and into variations of Covington's parsing strategy[25] on the other hand. Nivre's algorithm use two stacks, one for partially processed tokens and one with still to be processed tokens. There are then several possible operations, which either combine the tokens from the stacks with one another, move them from the unprocessed stack to the partially processed or remove processed tokens. Nivre's algorithms have linear complexity, because they are able to remove already processed tokens from the stack and these tokens do not interfere with other words during the later processing. Covington's strategy successively compares one token to all tokens left to it for all words in the sentence and for

every word pair decides whether there is a dependency relation between them or not. Covington's parsing strategy has quadratic complexity, because there is no possibility to ignore already processed words and in the worst case one word has to be examined together with all other in the sentence.

The graph-based strategies are first of all differentiated into first-order and second-order ones, where second-order parsers make choices for an edge depending on already chosen edges, whereas first-order parsers do not. As already mentioned, according to the graph-based approach one initially creates edges between all words of the graph before computing the dependency tree. There are techniques to avoid some of these edges, because they do not occur in the training data. Therefore word forms, POS tags and dependency labels can be taken into account in order to judge whether an edge should be considered at all, when the global solution is computed. As far as the computation itself is concerned most systems use the maximum spanning tree (MST)[53] as the best dependency tree, which can be computed with Chu-Liu-Edmonds algorithm[23]. The idea of this algorithm is to select all edges with the highest scores. If the result is a tree, then it terminates. Otherwise the result is still a graph with a cycle in it, so the cycle is removed by contracting the nodes involved in it into new artificial node. This is repeated until no cycles are left. After finding an MST on the contracted graph, one can reconstruct an MST for the original one out of the history of contractions[34].

Distributedness In the second half of the 2000s we have seen that the trend in processor development has moved to multi-core CPUs. At first the software and programming languages reacted slowly to the new trend, but at the present day parallel computing can no longer be ignored. Ordinary desktop or even mobile computers usually have 2-4 cores and we will see devices with 8-16 or more cores very soon (cf. hyperthreading[51]). Additionally, technologies like MapReduce[29] extend the possibilities of parallel computing to unlimited number of computers, thus allowing to use a lot of processors even if the individual devices are not very powerful. Parsing is an extremely suitable task for parallel computing: one sentence can be processed independently of the other and even within one sentence a lot of processing steps can be done in parallel. E.g. Feature extraction can also be parallelised, since feature values are also independent from each another[12]. In this work Bohnet reports 1.9 faster feature extraction with 2 and 3.6 faster extraction with 4 cores. Nevertheless most parsers still do not support parallel computing and only make use of one core.

Architecture Dependency parsing is a complex task which can be divided into many subtasks. Two major tasks are the creation of a head-modifier tree skeleton on the one hand and the labeling of the individual dependencies with their types on the other hand. Both can be then arbitrary divided into further subtasks. E.g. one subtask is to find all dependencies going left to right and another one to find those going right to left. One can differentiate between local and long-distance dependencies and between projective and non-projective ones. As we will later see for the already mentioned splitting technique one can even differentiate between dependencies for words of different parts of speech. The architecture of a system is thus a very important point and almost all parsers are different in this respect. MST Parser for instance does parsing and labeling independently of each other, whereas MaltParser can do those two tasks in one step.

Feature Models A central problem in machine learning tasks is the feature engineering part. One has to find a feature model, which works best with the machine learning approach one has chosen. The amount of work and the methods vary greatly for different classification approaches. For some approaches it is the job of the machine to find a reliable accurate model out of a big space of possibilities. Such approaches are very time intensive. For some other methods it is the job of the human developer to provide a good feature set, for which the classifier will learn a model. The training times are much shorter for these learning techniques, but still can take a considerable amount of time. With some machine learning approaches, in addition to finding good individual features, one also has to provide good feature combinations, in order to compensate for the missing feature interaction. For NLP tasks the number of features is usually very high[43] and it is computationally impossible to simply add all potential candidates that come to one's mind and let the classifier learn the correct figures.

Good features can significantly boost the accuracy of a parser and a compact set of features can guarantee good efficiency as well. However, most of the systems lack automatic feature engineering or feature selection techniques. Therefore they are usually slow, because the models include a lot of not so useful features, and an optimal accuracy is usually achieved only after a very long development period. As an example, MaltParser's accuracy for English with the very same algorithm and machine learning library LibSVM changed from 85.81 LAS[2] in 2007 to up to 90.4 in 2012. Some newer machine learning algorithms,

[2]taken from http://maltparser.org/conll/conll07/, last retrieved on March 23,2012

however, implicitly perform feature selection and produce much more compact models, which contain only highly useful features. Examples of such algorithms are L1 regularised logistic regression[85] or multi-class support vector machines by Singer and Crammer[42]. In regularisation a penalty term is applied to every feature weight, so that only the features with a big weight remain until the end of the learning. With support vector machines not all feature vectors are of equal importance. Only so called support vectors, i.e. instances of data which are involved into drawing the margin between the classes, are relevant and weights only for their features are learned.

Features can be of many different types. An important differentiation which is often done in the literature is whether the parser, and that basically means the feature model it is based on, is lexicalised or not. As the name suggests, lexicalised parsers learn a lexicon and a finite number of structures associated with the words from this lexicon. This is sometimes very helpful, because it helps to avoid many of the structural ambiguities. Lexicalised parsers therefore achieve better accuracies than those without a lexicon[44]. The drawback of lexicalised parsing is that the lexicon is big and there are thus many different features based on various word forms. Such models are harder to optimise and the efficiency of the system drops. On the other hand unlexicalised models are very compact, because they are based only on some general structural information, e.g. POS tags, are efficient and easy to optimise. Besides a lexicon there are many more other sources for features. One can use morphological information, subcategorisation and other databases, e.g. containing semantic information. The other important type of features are structural features. As with the already mentioned first-order and second-order parsing algorithms, one can define features which can use the partially constructed structure at any stage of the processing. These features might use different information about dependents, governors, siblings and neighbouring words or as well distance information between tokens of the structure constructed so far.

Projectivity In section 2.1.2 I have already defined the projectivity constraint for dependency structures. In many languages most structures satisfy this constraint and even if they do not, it is mainly due to one non-projective dependency. Non-projective dependencies usually arise due to irregular word order, which does not cause ungrammatical constructions in many languages, e.g. Slavic ones, but still is not the preferred variant. Thus the average number of non-projective dependencies is rather low, e.g. one in Czech, which is par-

ticularly famous for non-projectivity, the overall number of such dependencies is 1.9%[36] of the total number of dependencies. Considering their rareness it is often sensible to restrict oneself only to projective ones without losing much accuracy, but gaining a lot of efficiency. Therefore many parsers are either only projective or allow non-projectivity only as an option. Fully non-projective algorithms, i.e. those which always assume non-projectivity in the sentence they are parsing, are much more inefficient, since many more word-pairs have to be investigated, since basically arbitrary word order is allowed. Therefore there are some intelligent techniques, which allow more efficient processing of non-projective structures. First approach is called pseudo-projective parsing[68] and processes the sentence as if it were projective, but then recovers non-projective dependencies in a post-processing step. The second option is again to assume projective structures, but then to reorder words during the processing, if necessary, in order to get the desired non-projective structures[64]. The last option is based on the idea that the default strategy is to assume projectivity, but if there is an indication that non-projectivity might be involved the parser switches to the non-projective model[21].

Deepness This concerns the detailedness of the dependency analysis returned by the parser. The usual practice is to compute only a layer of syntactic analysis. However, there also some parsers which are able to deliver morphological analysis and/or a layer of semantic dependencies. The syntactic analysis usually makes use of some tagset in order to provide types for predicted dependencies. There are different kinds of tagsets. E.g. for English the most prominent are the CoNLL tagset[76] and Stanford tagset[28]. Both tagsets contain around 50 different types each and allow a quite rich annotation. For other languages there also exist tagsets with similar number and type of dependency relations. As far as semantic dependencies are concerned there are also some tagsets based on semantic roles. The data used in the CoNLL 2008 shared task for semantic role labelling was annotated using PropBank[71] and NomBank[55], which were used to annotate verbs and nouns, respectively. According to these tagsets the semantic arguments are simply numbered A0,A1,A2 etc. based on the entries in the according frames in these frameworks. Some other frameworks, e.g. FrameNet[7] allow a more detailed semantic annotations with meaningful labels like agent, patient, goal etc. Depending on how many layers of annotation a parser is able to produce, one speaks of shallow and deep parsers. Most parsers, including MST Parser, MaltParser, Minipar, Ensemble and Stanford Parser are

shallow and produce only syntactic dependencies. The projects called mate-tools and clear parser offer both a parser[12][21] and a semantic role labeller[11][22].

2.4 Summary

So far I have introduced the formalism of dependency grammar, explained how one can produce the respective representations by means of dependency parsing and listed the most important properties of such systems. For each property I have mentioned the state of the art parsers associated with it. I will briefly summarise the most important systems according to their most prominent characteristics before introducing my own parser MDParser in the next section. I will also assess some of their properties in a qualitative manner, however, a detailed quantitative evaluation will come in a later section dedicated to that.

MaltParser [65] is a data-driven transition-based deterministic parser. It provides various parsing strategies which support both projective and non-projective parsing and the grade of incrementality, i.e. the size of the lookahead, can be specified with a formal feature specification language. It also supports both parsing and labeling in one step, as well as doing it separately. It does not have a mechanism for automatic feature engineering, therefore the feature models have to be carefully specified and have a big impact on the quality of the results. There are currently two versions of machine learning strategies supported: LibSVM and Liblinear. I will refer to these as MaltParser(LibSVM) and MaltParser(Liblinear) in the course of the thesis. LibSVM requires a lot of time both during the training and testing phase, whereas Liblinear is very fast during both. MaltParser(LibSVM), however, produces slightly more accurate results than MaltParser(LibLinear). MaltParser can be trained for both Stanford and CoNLL dependencies and it only delivers the syntactic analysis of the input.

MSTParser [53] is a graph-based parser. Due to this fact it is non-deterministic and non-incremental by nature. It does parsing and labeling separately. The feature models are responsible for a great portion of its accuracy and therefore a careful feature engineering is necessary. MST Parser has a very complex machine learning approach behind it, which requires a lot of time and resources to train the model. The resulting model is huge and it also takes a lot of time and resources to apply in the parsing phase. MST Parser can learn both Stanford

and CoNLL dependencies and also produces only the syntactic analysis.

Stanford Parser [44] is a phrase structure grammar parser. It is able to derive so called Stanford dependencies out of phrase structures with help of head rules, which identify the head and the children for all possible PSG constructions. Since Stanford Parser is a PSG parser it has a completely different training phase. In PSG parsing one has to first read off all possible rules from the training treebank and in the second stage learn their probabilities. The starting point for that in Stanford Parser is the maximum-likelihood estimation and then some other steps like markovisation or smoothing[44] are performed to deal with the problem of sparsity. Stanford models are relatively compact, but because of the two stage processing (first PSG, then transformation to DG) the overall parsing time is still long. Stanford Parser is the only system, which is able to work with plain text. It contains all preprocessing components like sentence splitter, tokeniser, pos-tagger, which are necessary to transform the plain text input into a parsable format.

Minipar [49] is a rule-based dependency parser for English. It is an old parser which has not been developed for a long time, but it still popular because of its efficiency. Minipar is a chart parser which constructs all possible parse trees for a sentence and then selects the best. Additionally, Minipar's parsing algorithm requires all words to be initiated with their categories before the processing and thus the parser is neither deterministic nor incremental. As far as the quality of results is concerned Minipar performs worse than the state-of-the-art data-driven parsers. However, because it is rule-based it is able to judge whether a sentence is grammatical or not, whereas data-driven parsers always provide some dependency analysis for whatever input sentence given.

Ensemble [77] is a combination of all parsing algorithms implemented in MaltParser: Nivre's arc-eager, Nivre's arc-standard and Covington's non-projective model, each with the parsing direction left to right and right to left, plus the default algorithm of MST Parser: Overall 7 different models. Each individual parser runs in a separate thread which allows to parse at the time of a single parser, provided the necessary processing resources. Ensemble is different to many other experiments of combining different strategies at the learning time, so that one gets either only one hybrid model or the individual models at least are enriched with features based on the output of other models. Ensemble performs

the combination after parsing is done, by a voting mechanism. Additionally, it has a procedure to guarantee that the resulting dependency tree is well-formed. Otherwise the system has the same properties as MaltParser.

Clear Parser [21] is a new parser inspired by MaltParser. It proposes some modifications which improve both efficiency and accuracy. The efficiency improvement comes from a modification of Nivre's parsing strategy. Clear Parser differentiates between projective and non-projective structures and is able to avoid unnecessary search for non-projectivity if its model predicts so. For the accuracy boost the developers use a different training strategy. They do not only learn on the gold standard training data, but they also produce a variant of the training data parsed by the parser. Thus the parser is trained both on gold-standard annotation and automatically parsed trees. This helps them to create a better model, which allows to reduce the gap between the training data and what the parser is able to replicate. Otherwise the parser has the same properties as MaltParser. Besides syntactic analysis there is also a semantic labelling component.

Mate-tools [12] is another new parser, however, inspired by MST Parser. It proposes a modification for considerably improving parsing speed by means of optimising the feature extraction procedure, which accounts for the most time spent on processing. On the one hand the developers employ a method improving the mapping of feature values to their indexes, i.e. corresponding unique integers. On the other hand they make use of several CPU cores, if available, in order to extract features in parallel. Mate-tools also offers a semantic labelling component.

In this summary I have presented the most prominent parsers of the last years. The older data-driven parsers have had high accuracies as their top priority and are very slow. The only exception to this is Minipar, which is remarkably fast, considering the year of its development. Despite the fact, that it was less accurate it still enjoys a wide popularity. The probably most widely used system, however, is Stanford Parser, despite the fact that it is neither the most accurate one nor does it allow fast processing. In fact Stanford Parser is very slow, because it first creates phrase structures and it has problems with longer sentences. In my mind the popularity of the parser arises from the fact that it allows processing of plain text and the user does not have to search for components which have to do that in the desired format, as it is the case with all

other systems. The newer developments show that parsing speed is becoming more important, however the speed improvements are only admissible if they do not impair accuracy, as it is the case with more efficient data structures or parallel computing.

3 MDParser

I have presented an overview of the field of dependency parsing, explained the most important properties and described the most prominent systems which are available for the research community. In this section I am going to motivate the development of yet another parser MDParser. MDParser is supposed to be an application-oriented parser. I will discuss why it was necessary to develop a new parser and why the other ones are less suitable for this goal.

In the introduction (Section 1) I have explained that parsing is often seen as an independent NLP task. The usual setting is a gold-standard treebank, i.e. a ready-to-parse resource, consisting out of a well-formed grammatical tokenised text with known sentence boundaries and annotated with parts of speech. The goal is to replicate as much of the gold standard annotation as possible for some small portion of the treebank, which was held out. Therefore it is understandable that all systems are biased towards accuracy for this specific kind of task and data, while other properties are often neglected. However, when developing applications, i.e. natural language processing tasks which profit from dependency analyses of the data, the setting is absolutely different. The text is often not well-formed, since extracted from the web in form of HTML or PDF, all preprocessing steps like sentence boundary detection, tokenisation and part of speech are usually difficult, both because of ill-formedness and due to the fact that the data might be different from what these tools were trained on. It is therefore essential that the parser does not get misled by an input it does not expect. Additionally, the size of the data is always much bigger than the few hundreds of sentences in the development and test sets of the treebanks and therefore efficiency is very important. At last, since applications often process the web, multilinguality is also an issue. Therefore it is beneficial if the parser does not require much data for training to achieve decent results, because for many languages annotated resources, such as treebanks, are small. On the other hand applications rarely require full dependency analyses for their needs and often focus on some partial structures and thus the global accuracy for all kinds of relations is not as important as the performance on some specific dependency types.

Let us sum up the requirements for an application-oriented parser:

- Preprocessing of the data is important and should be tailored for the parser so it gets exactly the kind of input it was trained on.

- Efficiency is a major issue, because parsing is required for large amounts of data.
- The size of training data the parser requires should not be too big.

The only well-known state-of-the-art parser which does all the preprocessing steps is Stanford Parser. However, it is not efficient enough to work with a lot of data. In fact almost none of the parsers is fast enough to work with the web. The solution of newer systems to split the task into several threads and do the computation in parallel is very fruitful, but speeding up the processing by the brute computational power is not elegant and would lead to very unpleasant properties in case the computational resources are not available. Therefore I have decided to create a very fast parser, which is able to work with large amounts of data, is capable to do all the preprocessing and can be easily used for many languages. As far as the other properties are concerned, it should not be inferior to other systems.

When I started the work on this thesis and studied the state of the art, in order to get an impression what could be the starting point of my development, the last three parsers from my overview, as well as MaltParser(Liblinear), had not been yet available. Minipar is a rule-based system and Stanford Parser is a PSG parser, therefore they did not qualify. Thus the choice was either to base the new parser on MaltParser(SVM) or MST Parser. Both parsers were very slow and required hours to train models and minutes to apply them. The low speed of MST Parser lied in its graph-based nature, since it is costly to search for an optimal global solution in a non-deterministic and non-incremental way. Additionally, it required second-order parsing, i.e. information about already selected edges, in order to achieve good results. The low speed of MaltParser could be explained with the choice of its machine learning strategy, SVMs with a quadratic kernel. Otherwise its properties looked very promising: the decisions are local and the parsing algorithms are deterministic and require only a small lookahead. Therefore the decision was to build a transition-based system, which has all the above application-oriented properties, but is otherwise not inferior to MaltParser.

3.1 Framework for Dependency Parsing

In this subsection I am going to describe the architecture of MDParser, as well as point out and discuss its differences with MaltParser. Usually system

descriptions convey the general idea and do not contain all the technical details. That is why it is very difficult to make sure that some component is implemented exactly the same way as it is described somewhere in the literature. This is also true for MaltParser. Despite the extensive literature on this system, I was not able to find out many of the implementational decisions. Some of them I could recover directly from the code, however, many others remained unclear and required me to come up with my own solutions. Since MDParser proved to be able to parse sentences within microseconds, the implementational details become as important as the theoretical description of the system. Therefore I will always provide as many technical details as possible.

3.1.1 General Architecture

First, the parser has to read in the data it is going to work with. Then there are two possibilities, depending whether the system is run in the training or parsing mode. In case of training the system has to construct the training data and eventually learn a model out of it. In case of parsing the system has to construct the dependency analysis for every sentence in the data. I will describe both phases separately.

Training Phase In the training phase the system runs a parsing algorithm on the data using an oracle, a component which uses the gold-standard annotation to select the correct decision in each state of the processing. Each state of the algorithm corresponds to the partially built structure so far, the desired gold-standard structure and a pair of words, for which the system has to decide whether there is a dependency relation between them or not. First of all, the system checks whether the pair of words is permissible, i.e. if a dependency edge is introduced the structure is still a well-formed tree. If it is not permissible the oracle proceeds to the next pair of words, otherwise it selects the correct operation for this parser state. In the training phase it is very easy to do so, since one can consult the gold standard and check whether a particular relation for this pair is present or not. Otherwise the oracle selects some other operation, depending on the algorithm, in order to proceed to the next pair of words. For all permissible configurations the system applies the feature model and adds all extracted features into a feature vector together with the label of the correct operation for this configuration. Finally, a machine learning approach is applied in order to learn a model in order to select the correct operation in the parsing

phase, when the gold standard is not available.

Parsing Phase In the parsing phase the system runs a parsing algorithm on the data using the model learned during the training phase. In each state of the algorithm the model predicts what operation should be done next. The system also makes sure that no ill-formed structure (cf. Section 2.1.2) is constructed. It is easy to guarantee that for most of the constraints, so that e.g. no tokens get several heads, no cycles are introduced and in some cases non-projective edges are not allowed. However, with many algorithms some tokens might not get a head assigned and thus a post-processing step is necessary for them, in order to make sure that the final result is a well-formed dependency tree, i.e. the connectedness and single-headness constraints are satisfied.

3.1.2 Implementational Details

In this section I am going to provide the technical details of some important components of MDParser and discuss the differences between my implementation and the one of MaltParser.

Oracle is a component, which looks up the correct decision for the parsing algorithm during the training phase. Usually algorithms have between three and six possible operations to choose from. Whereas it is easy to look up, whether for a given pair of words j and i there is a gold-standard dependency (j, i) or (i, j), it is sometimes not straightforward which operations to choose in case there is no dependency relation between two given words. E.g. Nivre's arc-eager algorithm has two operations for this case: SHIFT and REDUCE. With SHIFT the parser moves to a different pair of words, while the previous indexes are still kept on the stacks, i.e. they are considered *partially* processed and might become involved in the dependency structure later. With REDUCE one of the words is removed from the stacks, since it is considered being already *completely* processed, i.e. it already has a head and the oracle knows that it will not be a dependent for any of the forthcoming words. A similar situation arises with the version of Covington's algorithm implemented in MaltParser. It also has two possible operations in case there is no dependency for the words j and i. One operation is called NO-ARC and it is applied in order to go over to next i, i.e. $i\text{-}1$ in order to eventually find a head or dependent for j. The other one is called SHIFT and is supposed to change j to $j{+}1$, because the previous j will

not be involved in any dependency relation with the possible i's at that stage of processing.

First, it is unclear which one to select first. Second, there are no descriptions of how the procedure for checking whether a token is partially or completely processed looks like. For the first issue there are basically two options: a) prefer REDUCE, i.e. reduce a token as soon as possible when it becomes completely processed and b) prefer SHIFT, i.e. reduce a token only when necessary in order to build the desired result and it is otherwise interfering with the processing. I have chosen the first option, which decreases the overall number of transitions necessary to process a sentence. The drawback is that one has to check whether a token is completely processed every time in order not to accidentally remove a token which is still involved in the processing. With SHIFT first, one does not have to worry about it, but the overall number of states might increase. It is very similar for the Covington's algorithm with the two operations SHIFT and NO-ARC.

The procedure of checking whether a token has been processed or not depends on the algorithm. For Nivre's algorithm it is quite easy. Given a token j, which is potentially to be reduced, one has to check whether it already has a head. If not, then it has not been completely processed. Otherwise, one has to check all other not completely processed tokens, whether they have the token j as their head. If not, the token can be safely reduced, otherwise one has to keep it. With Covington's algorithm it is a little bit more difficult. The algorithm checks for every word j all words i_n left to it whether there is a dependency relation between them. That means that one can not shift to the next word *j+1*, unless there is at least one word i, so that there is a dependency (j, i) or (i, j). That means that one has to first check whether j has a dependent left to it and if yes, then shift is not allowed. Otherwise one has to check for all i's to the left of j whether they have j as a parent. Only if it is not the case, then one can safely proceed with *j+1*.

The latter implementation is probably not as efficient as the one in MaltParser, which internally uses chart parser like data structures and does not spend so much time on these checks. However, the oracle is only used during the training phase and thus does not influence the parsing speed, so it has not been optimised too much.

Parsing Algorithm In transition-based parsing the goal of the parser is to find all pairs of words (j, i) for which the dependency relation holds. The

Algorithmus 1 Covington's algorithm

```
1 for (int j=1; j < sentence.length;j++) {
2   for (int i=j-1; i >= 0; i--) {
3     if (permissible(j,i) || permissible (i,j) {
4       select correct transition
5     }
6   }
7 }
```

naive strategy would be to examine all possible word pairs in the sentence. This is inefficient since many of the pairs are impossible, e.g. because of the ill-formedness constraints. Therefore there are more clever ways of restricting the search space without missing any of the desired relations. I have already presented some of the most popular ones, like Covington's or Nivre's algorithms.

MaltParser contains implementations of almost all prominent algorithms. During the development of MDParser I have also tried many of them and they are still part of the system, however, I have found out that Covington's algorithm has particularly appealing properties and thus it is the MDParser's default parsing strategy. There is not much to say about the implementation, since Covington's algorithm basically requires two simple for-loops to go through all desired word pairs and it is implemented the same way as in MaltParser.

Permissibility Before consulting an oracle or a model, which transition is the correct one in a certain configuration, it is sensible to examine first, whether the given pair of words would violate the well-formedness constraints for some of the possible operations. Therefore there is a procedure to check all constraints defined in the sections 2.1.1 and 2.1.2. Unfortunately there is no description of how they are implemented in MaltParser and thus I had to provide my own implementations. Permissibility check (line 3; Algorithm 1) is a quite important part of the algorithm, since checks for projectivity (Algorithm 2) and acyclicity (Algorithm 3) are quite complex procedures and have to be performed millions of times during parsing, even for relatively small data sets. I will provide my implementation only of these two, since the other ones such as single-headiness or anti-reflexiveness are self-evident.

The algorithm 2 checks whether the edge between two words with indexes j and i is projective or not. An edge of length 1 is always projective (lines 1-3). Otherwise one has to check which token has a smaller index (lines 3-9). Let us assume i is the smaller and j is the bigger one. Then for every word between

Algorithmus 2 Projectivity check

```
1 if (distance(j,i) == 1) {
2   return true;
3 } else {
4   int smaller = j;
5   int bigger = i;
6   if (j > i) {
7     smaller = i;
8     bigger = j;
9   }
10  for (int i=bigger-1; i > 0; i–) {
11    if (i != smaller) {
12      if (i > smaller && head(i) < smaller) {
13        return false;
14      } else if (i < smaller && head(i) > smaller) {
15          return false;
16      }
17    }
18  }
19 }
20 return true;
```

i and j (line 10), there should be no word which has a parent which is smaller than i (lines 12-14) and for every word between 0 and i there should be no word which has a head which is bigger than i (lines 14-16).

The algorithm 3 goes through all tokens, for which holds that i is their subordinate, i.e. i is reachable in the tree by following down the dependency edges. It starts with the current head of i (line 5) and then its head (line 15) etc. until it reaches the artificial node 0 (lines 9-11). If any token comes on the stack for the second time, we know there is a cycle and terminate (lines 12-14). Otherwise if the token 0 can be reached, then we know that there is no cycle on the path.

Features In machine learning a feature is a property of a phenomenon being observed. A good discriminating feature occurs often with one class and rarely with the other ones, so that it becomes an indicator for a certain outcome. Features in transition-based parsing are extracted by applying a set of feature templates in every permissible configuration. Feature templates are functions, e.g. $pos(j)$, which would return the part-of-speech tag of the word j, or $wf(j)$, which would return the word form of the word j. The encoding

Algorithmus 3 Acyclicity check

```
1  int curHead = j;
2  boolean[] possibleEnds = new boolean[sentence.length+1];
3  possibleEnds[i] = true;
4  Stack<Integer> toCheck = new Stack<Integer>();
5  toCheck.add(curHead);
6  while (!toCheck.isEmpty()) {
7    curHead = toCheck.pop();
8    int curHeadHead = heads[curHead];
9    if (curHeadHead == 0) {
10     return true;
11   }
12  if (possibleEnds[curHeadHead]) {
13    return false;
14  } else {
15      toCheck.add(curHeadHead);
16      possibleEnds[curHead] = true;
17   }
18 }
19 return true;
```

of the corresponding features might vary, but usually the values are strings, e.g. a part-of-speech tag of a certain word (DT,NN,VB etc.), a word form (the, economy,be) or a dependency label (SBJ, OBJ, ROOT). In order to keep track which feature template is responsible for which feature the final value in the training data usually contains indication for both, e.g. "pos(j)=DT".

In addition to such easy-to-understand feature templates, which have a clear semantics, there are also artificial feature conjunctions, which are concatenations of some original feature templates. An example for a feature conjunction would be "pos(j)=DT#wf(j)=the". Feature conjunctions are a means in order to compensate for lack of exploration of feature space with non-kernel-based machine learning approaches (cf. the machine learning paragraph in Section 2.3).

We can observe that the extraction of features involves a lot of string operations. First a feature has to be constructed out of the feature template's name plus the equality sign plus the feature's value. For feature conjunctions the individual features have to be constructed and then their values have to be concatenated with the hash symbol[3]. Since for security reasons String is an immutable basic type in Java (this is a problem specific for a particular programming language, but all other parsers are also in Java and most machine learning

[3]It could be an arbitrary symbol not occurring in the data.

libraries are also available only in few languages, Java being the most common), each time you append something a new String is created, the old value is stored the new value is added, and the old String is thrown away. The longer the strings the longer the concatenations take, but even for typical feature lengths of about ten characters a concatenation takes around 0.25-0.3 microseconds with the fastest way of concatenation in Java. I have tried out three possibilities of concatenation: using '+' operator (corresponds to String.concat()), using StringBuilder class and using String.format() method. The '+' operator was the fastest one for me in this scenario. In MaltParser StringBuilder is used.

There is one more important property of features. Some features are the same for many configurations, because they do not change in the course of processing, as e.g. word forms or part-of-speech tags. On the other hand some other features, e.g. based on the partially constructed structure, as e.g. dependency labels of some tokens, do change in the course of processing, usually from being *null* in the beginning to some value at a certain point of parsing progress. I will call the first class static and the second class dynamic features. Since I have already demonstrated the costs of string operations for features, it is sensible to avoid constructing static features all over again. I will propose a method for this in the following section of the thesis.

Alphabet I have explained that features on their surface form are strings. However, internally machine learning libraries always work with integers. Sometimes the library itself has the functionality to index all features and assign each possible value a unique integer and sometimes they expect the input to already be in the integer format. The packages LibSVM[20] and LibLinear[33] are of the latter class.

A mapping between string feature values and integers is called alphabet. Both MaltParser and MDParser use Java HashMap class in order to implement this structure. However, whereas I have used the default Sun implementation of this class MaltParser uses a specialised Google implementation, which allows a particularly fast retrieval. In addition to that, I use one flat HashMap for the entire mapping and MaltParser uses a complex HashMap cascade, where all features are grouped according to their values, so that e.g. word forms are looked up in one HashMap and POS tags are stored in a different one. The idea is that the majority of the features, which are POS-based, require as little time as possible, which is the case when the HashMap is smaller. I have not experimented with different implementations of the alphabet, however, my

analysis of the processing times has shown, that it is not a very costly part of the parsing process and accounts only to few percents of the entire time spend on the processing and thus its optimisation is not very important.

Models During the learning, the features get weights for each class, determining how indicative a certain feature is for every possible class. A positive value increases the probability of a certain class and a negative value decreases it. The higher the value, the bigger the effect on the probability distribution. The learned feature weights are then printed into a file after the training is done. For parsing they are read in into a two dimensional double-array: one dimension for each feature and one dimension for each class.

Many learning algorithms, such as L1R-LR or MCSVM_CS from the LibLinear library end up with zero weights for many features, since the features either get regularised by a penalty term or they do not belong to support vectors and are thus not considered. In fact, only few percents of all features do get a non-zero weight in the end. It is therefore sensible to eliminate those in order to make the array with the weights smaller, both for higher access times and lower memory requirement. LibLinear does not support such compaction and thus the functionality has to be implemented manually.

MaltParser's compaction includes removing features with zero weights, as well as mapping features with the same weights to one entry. MDParser only removes those with zero weights, since the additional mapping to equivalence classes decreases efficiency. Eventually, the alphabet has to be adopted, since the indexes of features have changed and many of the original features in the alphabet were removed. Again, as with the alphabet data structure, I have measured the costs of accessing the weights array during the parsing phase and have found out that it is not very costly. In fact retrieving objects from arrays is very cheap in Java and does not account even for 1% of the entire execution time, despite the fact that it is done millions of times. Therefore I have not put a lot of effort in its compaction.

Feature Vectors For every configuration a lot of features have to be extracted. These features are then put together in some kind of a data structure. This collection is usually called feature vector, however, the underlying data structure should not be necessarily a vector. In fact, a vector is not the best data structure for parsing purposes, since it only allows efficient access to its elements, but adding new elements is expensive. Even though the number of

feature templates is constant, the exact number of features for a particular configuration is still not known beforehand, because some of the values returned by the templates will have zero weights and thus will not be included into the feature vector. Therefore a dynamically expandable collection, such as a list (e.g. the usual Java ArrayList class), is a better option.

For the training phase the LibLinear package has an additional requirement: all features in the vector have be sorted in ascending order according to their indexes. Thus the data structure for the feature vector actually has to be a sorted list. It is important to note, that the parsing phase does not have the latter requirement and it is thus very important to differentiate between feature vector types depending on the phase. Sorting is a very expensive procedure and accounts for a big portion of execution time. It is important to choose the most efficient sorting algorithm, such as quicksort implemented in the Java Collection class. Using less efficient algorithms such as insertion sort degrades the efficiency by multiple times.

3.2 Feature Model

The performance of a transition-based parser relies heavily on its ability to predict correct transitions. In this subsection I am going to list all features I have used in MDParser for this task. These features are only based on word forms, POS tags and dependency labels, because this information is available in any dependency treebank for any language. Some languages contain additional information about lemmas or different morphological knowledge, but I did not exploit it, since it is not available for every language and it also would require additional preprocessing components, which are also not easy to get.

1. WFJ $\Rightarrow$ returns the word form of the token j.
2. PJ $\Rightarrow$ returns the part of speech of the token j.
3. WFJP1 $\Rightarrow$ returns the word form of the token $j+1$.
4. PJP1 $\Rightarrow$ returns the part of speech of the token $j+1$.
5. WFJP2 $\Rightarrow$ returns the word form of the token $j+2$.
6. PJP2 $\Rightarrow$ returns the part of speech of the token $j+2$.
7. WFJP3 $\Rightarrow$ returns the word form of the token$j+3$.

8. PJP3 $\Rightarrow$ returns the part of speech of the token $j+3$.

9. WFI $\Rightarrow$ returns the word form of the token i.

10. PI $\Rightarrow$ returns the part of speech of the token i.

11. PIP1 $\Rightarrow$ returns the part of speech of the token $i+1$.

12. WFHI $\Rightarrow$ returns the word form of the head of the token i.

13. PHI $\Rightarrow$ returns the part of speech of the head of the token i.

14. DEPI $\Rightarrow$ returns the dependency label of the head of the token i.

15. DEPLDI $\Rightarrow$ returns the dependency label of the left-most dependent of the token i.

16. DEPRDI $\Rightarrow$ returns the dependency label of the right-most dependent of the token i.

17. DEPLDJ $\Rightarrow$ returns the dependency label of the left-most dependent of the token j.

18. DIST $\Rightarrow$ returns the distance between the tokens j and i. For $i = 0$ the feature returns 0, for the distance 1 the feature returns 1, for distances 2 or 3 the feature returns 2, for distances 4 or 5 the value 3 is returned, for distances 6, 7, 8 or 9 the value 4 and for all other distances the value 5 is returned.

19. MERGE2(PI,PIP1)$\Rightarrow$ returns the concatenation of PI and PIP1 features.

20. MERGE2(WFI,PI)$\Rightarrow$ returns the concatenation of WFI and PI features.

21. MERGE3(PJP1,PJP2,PJP3)$\Rightarrow$ returns the concatenation of PJP1, PJP2 and PJP3 features.

22. MERGE2(DEPLDJ,PJ)$\Rightarrow$ returns the concatenation of DEPLDJ and PJ features.

23. MERGE3(PI,DEPRDI,DEPLDI)$\Rightarrow$ returns the concatenation of PI, DEPRDI and DEPLDI features.

24. MERGE2(DEPI,WFHI)$\Rightarrow$ returns the concatenation of DEPI and WFHI features.

25. MERGE3(PHI,PJP1,PIP1)⇒ returns the concatenation of PHI, PJP1 and PIP1 features.

26. MERGE3(WFJ,WFI,PJP3)⇒ returns the concatenation of WFJ, WFI and PJP3 features.

27. MERGE3(DIST,PJ,WFJP1)⇒ returns the concatenation of DIST, PJ and PJP1 features.

Feature templates 1-13 are lexical (word forms and POS tags), 14-17 are based on dependency labels and template 18 takes the distance between two words into account. Templates 19-27 combine the templates 1-18 between each other.

This is a model with a lookahead of 3, which uses a lot of static features templates (all word form templates and all POS templates are static by default). Whereas most templates are identical to those used in MaltParser (1-18 are taken from the MaltParser model), I have modified some feature conjunctions, so that their values can be reused as often as possible (e.g. templates 19-21 are static, whereas MaltParser did not contain static conjunctions at all). In the next section I will provide more details on feature reusability.

4 Contributions

So far I have introduced the dependency grammar formalism, explained the notion of dependency parsing and described my implementation of MDParser and its differences to MaltParser, on which it is based. In this final section of Part I I will present my experiments in the field of dependency parsing with the ultimate goal of creating a particularly suitable parser for applications.

The experiments can be divided into two groups: those which failed and those which succeeded. Of course the latter are more interesting, however, I have also learned a lot from the experiments which did not succeed and therefore I will present both.

4.1 Trial and Error

In the introduction (Section 1) I have already briefly explained what kind of experiments I have done either without success or which became obsolete in the course of the years I have spent on this thesis. These experiments included the work on different machine learning packages, parsing strategies, feature usefulness, splitting and confidence values. In this subsection I am going to present them in details.

Machine Learning Packages In order to select the correct operation in each configuration one has to learn which features are indicative for which operation. For this task there are numerous machine learning approaches, which are able to learn adequate feature weights. Whereas they are often based on different approaches how this task is solved, the general idea is the same for all of them: learning feature weights is seen as an optimization of a convex function. The general procedure then looks as follows:

1. Initialise with some value, e.g. 0
2. Evaluate the weight (every method has some kind of an objective function, which reflects how many erroneous/correct predictions are made with the current weight)
3. Determine the direction of optimisation (since the function is convex, the optimum can be reached by either decreasing or increasing the initial weight)

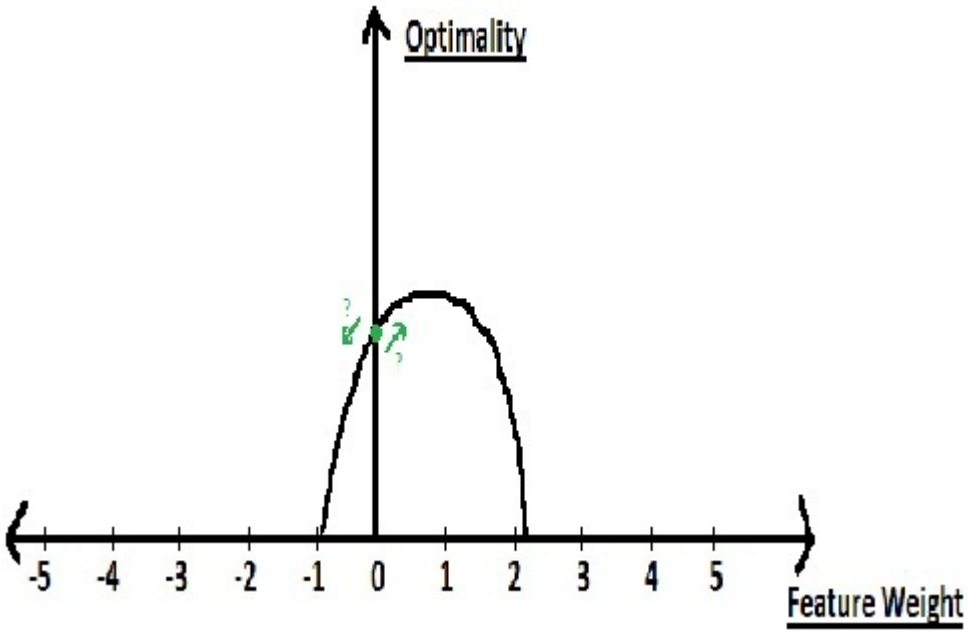

Figure 2: Convex optimality function for a feature

4. Step into the predicted direction (this is basically the main part of the optimisation problem, because the size of the step determines the number of iterations required to reach the maximum)

5. Repeat 2-4 until the maximum (i.e. the most optimal weight) is reached (since the function is convex, the maximum is the point where the gradient is 0).

The figure 2 visualises the optimality function:

The initial feature weight (marked by a green point) is 0 and it is not optimal. Decreasing the weight would cause even worse results, whereas increasing the weight at first improves it and after the angular point it begins to decrease. Therefore it is important to first determine the direction correctly, advance to the maximum in as few steps as possible and do not pass by it, since the performance will deteriorate if the weight becomes too large.

Considering this procedure every machine learning approach can be basically evaluated by the following properties:

- How many iterations are necessary in order to reach the maximum?

- How close to the maximum the algorithm terminates?
- For how many features should the procedure be done in order to achieve a good overall result?

The first machine learning approach I have tried was the maximum entropy classifier from the OpenNLP package. Its learning is based on the generalised iterative scaling[27] method. According to this method the stationary point of the optimality function is found by locally approximating the function as being linear. Since it is obviously not true (as one can see on the graph the function is definitely not linear, but rather hyperbolic) , and the gradient is not constant (as it is the case in linear functions) the approximation works only if the steps are very small. Additionally, the method is run for all features in the training data, without any feature selection techniques. Therefore a lot iterations are necessary in order to reach an optimal solution. Due to time constraints, however, the number of iterations is often limited, e.g. by 100, and therefore the learning terminates before the optimal solution is reached. Overall the approach worked relatively fast, compared to the non-linear machine learning approaches, which were usually used at that time. However, the results were much lower, because generalised iterative scaling could not approximate the optimality function well.

My next approach was again a maximum entropy classifier, however, with a different learning strategy. The strategy was called Broyden – Fletcher – Goldfarb – Shanno (L-BFGS) [14]. L-BFGS is a quite complex method and is difficult to implement, therefore I could find only one software package in Java, called Mallet, partially[4] supporting it. It is superior (faster and more accurate) to other optimisation strategies, such as Generalized Iterative Scaling[27] used in many other machine learning packages. Robert Malouf claimed that L-BFGS is the best optimization strategy for maximum entropy models in his comparison of different methods [50]. This strategy assumes that the function can be approximated quadratically, which is a better approximation and provides a better result. However, this method also optimises all given features and is therefore slow and terminates often before the optimal result is achieved.

The third algorithm I have tried was L1RLR (L1 regularised logistic regression)[85] from the LibLinear[33] package. Its main advantage compared to the previous two strategies is the regularisation. Regularisation is a technique to prevent overfitting by penalising the model by its complexity, i.e. the number of features. When regularisation is applied, the penalisation term steadily grows

[4] It does not work for problems with a large number of classes.

with the increasing number of features getting a non-zero weight and at a certain point only very good features, i.e. features with very big weights, are able to surpass the necessary threshold. This implicit feature selection helps to considerably reduce the number of features for which a weight has to be learned. This allowed me to use much larger feature models, which was too expensive with the maximum entropy classifiers I used before. The richer models, plus the fact that only a small percentage of weights had to be learned, resulted in very good results. However, due to the large initial training data size and quite complex computation behind this approach, it was not very fast.

Finally, I have experimented with the MCSVM_SC (multi-class support vector machines by Singer and Crammer)[42] from the same LibLinear package. It is different from most other approaches, including the three I have tried out before, because it is a real multi-class classifier. The usual practice is to train binary classifiers and if the problem has several classes, then one vs. one or one vs. rest strategies are applied in order to combine them. Additionally, it has similar properties as L1RLR in terms of the number of features for which a weight has to be learned. Since it is a technique using support vector machines, the weights are learned only for features in the support vectors, which are important for differentiating between various classes. Thus it allows very compact models, very fast training and good results.

Unfortunately, I can not present an overview how exactly the performance of the system evolved when the machine learning packages were changed, because the other components of the system were not constant neither, but also evolved in the course of the work. The increasing performance of the system had three reasons: a) the choice of the machine learning strategy had a direct effect on the increased performance, simply because the machine learning method was able to learn better weights, b) the change in the machine learning indrectly improved the results, because the new approach allowed more features to be included in the data or a more fine-grained set of operations and c) the performance increased because other properties like the parsing strategy or the architecture of the system changed, independently of the properties of the classifiers I was using. If it were only a) then such a comparison would make sense, because it would clearly demonstrate the difference of the ML approaches, however, because of significant changes in b) and c) the comparison is not possible.

Examples for changes of the b) class are that e.g. the number of operations in the parsing strategy was only three in the beginning (LEFT-ARC, RIGHT-ARC, SHIFT), because it would minimise the number of binary classifiers to be

learned. With the latest multi-class classifier I could add the fourth operation (NO-ARC), which had a positive effect on the performance of the system, because the number of states necessary for parsing could be reduced (cf. the Oracle section 3.1.2 for more details on different operations). Additionally, the richer feature models using dependency type features (cf. Section 3.2; features 14-17 in the feature model) had a positive effect only if the dependency types could be predicted with a sufficiently high accuracy, which was not the case for the first machine learning strategies I used, but worked fine with the latest. An example for the c) class is the efficient feature extraction strategy that I am going to present in the Section 4.2. It allowed a much better performance independently of the machine learning strategy.

In general, the accuracies (UAS) evolved from around 80% with the OpenNLP library, to 84% with Mallet, to 88% with logistic regression and finally 90% with linear support vector machines. The speed rose from few sentences per second with the maximum entropy libraries to around 14 sentences per second with logistic regression and to more than 1200 with support vector machines. However, as I have just discussed, the changes are not due to the machine learning alone, but also due to many other details. Especially, the final fast speed of MDParser is mainly due to the reusability of features and not the machine learning library itself, since MDParser outperforms MaltParser using the same machine learning library by a factor of 4. As far as the hardware requirements are concerned, machine learning usually required a special 64Bit machine with a lot of memory in order to run most of the parsers I have experimented with. MDParser also usually needed such a machine with at least 20-30 GB of memory for most of its models. However, in the end I have optimised the final version by compactising the models, i.e. by adjusting the indexes of features in such a way that they are as small as possible (e.g. if a feature 100 and a feature 105 have non-zero weights, but all features in between do not, then I would change the index of the feature 105 to 101) and by always printing the results of every single training step to the hard disk. This way there are a lot more Input/Output operations, but not so much information has to be stored in memory and therefore the final version could be run on an ordinary machine with 1 GB whereas all the previous ones could not. However, this is again not only due to the better properties of the machine learning library, but also due to the architecture of the system.

Lesson learned:

Despite the fact that I had to learn to use three machine learning approaches, implemented a lot of code to utilise them and conducted countless experiments

on optimising feature models for different languages, which in the end were not used, I have learned a lot from this work. The most important finding was that a very small percentage of features is responsible for almost the entire performance. Therefore it is decisive to have some kind of a feature selection methodology, because this way the training data is allowed to be bigger and one can still get much more compact models, since a lot of initial features will be filtered out.

Parsing Strategies Similarly to MaltParser I have implemented the support of the most prominent transition-based parsing algorithms: Covington's parsing strategy, Nivre's arc-eager and Nivre's arc-standard algorithms. I have then experimented a lot with the different strategies in order to obtain the optimal performance. During the experiments I have realised that the Covington's algorithm has particularly appealing features: it can be made projective/non-projective by changing a single parameter of the permissibility function (Nivre's algorithms are only projective and require additional techniques for non-projective structures, as e.g. pseudo-projective parsing), it is very easy to implement and modify, and it achieves very good accuracies for most languages (especially because it does not eliminate tokens from the agenda it is less error-prone to early mistakes, which is one of the main weaknesses of the transition-based approach). However, it is much less popular than Nivre's algorithms because it has quadratic complexity and Nivre's algorithms are linear. This leads to the fact that the parser requires more states to process the data, which is responsible for larger training data with more instances during the training phase and slower processing during the parsing phase.

I have invested a lot of effort in order to come up with a modified version, which has a reduced amount of necessary states. In the original Covington's algorithm the idea is that for every word j the parser examines all words i to the left of it in order to determine whether one of them is j's head or dependent. For the words in the right end of a sentence the number of the examined pairs can grow to a quite big value, despite the fact that for the overwhelming majority of these words there is absolutely no need to look for heads/dependents very far behind. In fact more than 90% (depending on the language the exact number varies, but the tendency is the same for all languages) are local, which means that the distance between a dependent and its head is small (usually not more than few tokens away), whereas the average sentence length is usually much higher than 20 (e.g. for English it is 25). One of the observations was that

all dependencies, except for one root word, occurred within one clause and if a sentence consisted out of several clauses it did not make sense to look for heads/modifiers outside of a clause. The algorithm I have come up with thus parsed all clauses independently of one another and then combined the individual sentence parts together, determining which clause is the main one and which ones are relative. The modification actually worked very well and could save a lot (25% on average and up to 50% for long sentences) of unnecessary pair examinations. However, this strategy introduced an additional error source: namely the detection of clause boundaries. Clause boundaries are always marked by punctuation signs, however, sometimes punctuation signs such as commas are also used for other purposes, e.g. enumeration. Therefore I had to train a model which was able to distinguish between these two classes of punctuation signs: those which marked clause boundaries and those which did not. I was able to distinguish between those two classes very reliably (accuracy higher than 98%). Furthermore, the remaining few percents which are classified incorrectly can be divided into two classes: false negatives (a punctuation mark is not classified as a clause boundary although it is one) and false positives (a punctuation mark is wrongly classified as a clause boundary). The first error class is harmless, since it just undoes the benefits of the strategy and makes the search space as large as it would be in the original Covington's algorithm, but it does not affect the accuracy. The only problematic case is the second one, since an incorrect clause boundary hinders the parser from searching in a certain part of the sentence, which can result in numerous erroneous decisions. Unfortunately, this error source therefore affected the accuracy of the parsing algorithm and the time saved by skipping unnecessary pairs was partially consumed by the prediction of clause boundaries and thus only provided a 10% efficiency boost.

In the end it turned out that a much better modification to reduce the number of transitions with the Covington's parsing strategy is the version implemented in MaltParser. Whereas the original strategy contained only one shift operation, which altered the index of i and moved over to the next j only after i was decremented to zero, the MaltParser's modified strategy contains an additional operation which can be applied in order to immediately proceed to the next j, independently of the i. The advantage is that no separate model is necessary and therefore this version is very efficient.

Lesson learned:

The experiments with parsing strategies have helped me to better understand two main properties of parsing algorithms: the number of states required to

parse a sentence and the number of features required to be extracted in each state. Even though a smaller amount of states means that the feature extraction has to performed less times as well, it does not help a lot if this reduction of the number of states is achieved by adding further models with their own features, which have to be extracted as well. The best possibility to improve the efficiency of an algorithm is thus to modify the inventory of available operations, so that less states are necessary to arrive at the final result and no additional features are required because the same model is used for processing.

Feature Usefulness Feature models are a major factor for both accuracy and efficiency of a parser. The more features are present in the training data the better the chance that the learner will find good discriminative features necessary for accurate predictions. However, the bigger the number of features the more intensive is the training and the slower the processing. The ideal scenario is thus that the training data should contain only a relatively small amount of very good features.

Usually the quality of a feature can be assessed only empirically. The traditional way is to train a model with and without a certain feature in order to measure its usefulness. Especially with the machine learning methods which were available few years ago, this is a very time consuming process. For a multilingual parser it becomes intractable to try to optimise the system for many languages under such circumstances. Therefore a possibility to estimate feature quality prior to training a model was highly desirable in order to achieve optimal results and efficiency.

I have come up with such strategy, which worked well for the already mentioned MaxEnt classifiers which did not contain any feature selection. The observation behind my strategy was that the usual typical features, like the word forms and POS tags of the potential head/modifier words, as well as of their surrounding context, are included in any feature model for any language and are usually useful. The real challenge is to find out which feature conjunctions, which are essential for any linear classifier, should be defined, i.e. which features should be combined with which features and how often. With the initial 20-30 ordinary features, there are already hundreds of such potential binary combinations and thousands if one also explores trinary ones. These combinations differ across languages and finding good ones empirically is a very time consuming task. The approach I have developed only required to train a model with the original primary features and the best combinations could be estimated only

with its help without having to train numerous models by adding them one by one.

The approach was to use the learned weights for the primary features in the learned model in order to compute which feature contributed how much to the success or failure of the classification task. A good feature should contribute a lot towards the correct class and greatly penalise the wrong ones. Thus one can go through all instances in the training data with the learned weights and compute the overall contribution of each individual feature. Additionally, I have come up with a metric for measuring the difficulty of finding the correct class by analysing the difference between the scores for different classes in each state. If the difference between the competing classes is big, then the classification task is easy, since apparently not many features favoured a wrong class and penalised the correct one. On the other hand a smaller gap between different scores indicates that many features did not have correct weights. The features that were useful in easy cases are not that important because all other features were good as well. On the other hand the features which helped in the difficult cases should be prioritised and the ones which hindered the classifier by contributing to the wrong classes should be damped. Thus the score of every feature was also factorised by their behaviour in difficult cases. The result could then be used in order to rank the features according to their usefulness and the feature conjunctions could be then selected considering these values, i.e. by constructing feature combinations out of the most useful ones.

This approach, however, become obsolete with the newer machine learning strategies of the LibLinear package. First, the strategies already contained a methodology for feature selection so that only useful ones remained after training. Second, the training was much faster, sometimes only a matter of seconds, so that the traditional empirical feature engineering was again a reliable option. Third, no intermediate state, i.e. first training without and then with feature conjunctions, was necessary and the final model could be trained right away. Therefore, the final version of MDParser no longer relies on this methodology.

Lesson learned:

I have always seen the possibility to automatically estimate the quality of features in a non-empiric way as a central task in dependency parsing. For me it has therefore always been the top priority to address and solve this problem. Unfortunately, the approach I have come up with was not the most efficient and successful one and was finally outperformed by many others. However, I see it as an affirmation that it was a very important direction of research and

an essential property for a dependency parser to be able to select good features only.

Splitting Machine learning is usually a very computationally expensive task if the training data contains millions of instances and hundreds of thousands features, as it is the case for transition-based dependency parsing. Even if sufficient computational resources are available it then takes very long to actually train a model. Therefore it is sometimes sensible to reduce the size of the problem or to split it into many smaller ones. In MaltParser splitting is done in a such way that the training data is split according to the value of some feature, e.g. POS tag. This way one gets one model for verbs, one model for nouns, one model for adjectives etc. I have initially tried a different approach. The idea was to split the data purely by size, train the models, estimate the most useful features with the above-mentioned technique and then train a model for the whole data without splitting, but only with the estimated useful features. There were two reasons for why I did not want to use splitting for the final model. First, I was afraid that for many languages the treebank size is not sufficient even if the whole data is used and splitting can only deteriorate the result. Second, since there is no longer only one model, but many, for every classification one has to look up which model has to be used, which takes additional time. However, since feature usefulness estimation became obsolete, this way of splitting did not make sense any longer neither. Thus the final version of MDParser contains splitting as it is realised in MaltParser. Eventually, it also turned out that with the newer machine learning strategies already very small amounts of data are sufficient and therefore splitting works fine even for languages with small treebanks.

The MaltParser version of splitting has two parameters: the minimum size of one piece of training data and the feature according to which the data is split. The first parameter is necessary, because some feature values are so rare that it is not possible to train an accurate model for them. E.g. when splitting according to POS tags some tags like LS (List item marker), RP (Particle) or EX (Existential *there*) are not frequent enough and should not get a separate model. This parameter especially makes sense when splitting according to some other more diverse feature values. I have experimented with values of different feature templates, however, the one also used in MaltParser, namely PJ, worked best.

Lesson learned:

Since learning is expensive for big amounts of data, which is usually the case for dependency parsing, it was interesting to learn that a certain reduction of the training data hardly influences the quality of the results, however, immensely speeds up the processing. Unfortunately, my idea to work with small pieces of data and then carry over the findings to the whole data did not succeed for several reasons. First, the feature usefulness, as I was estimating it, could no longer be applied. Second, the strategy consisted out of two stages, namely training for the split data and training of the final model, which is less efficient than as it is in done in MaltParser, where the models learned out of the split data are already the final ones. As far as the additional time for choosing the right model is concerned, it is balanced out by the fact that one can access the weights in smaller models faster than one could do it in the one big model before.

Confidence Values Parser results are usually used in some follow-up applications as an additional source of linguistic information. In general the quality of the results should be at least as good that the application performs better using the dependency analyses than without using them. One can narrow down this quality constraint to individual dependencies: how likely is the predicted dependency good enough to be beneficial for the application or maybe is it better not to use the prediction because it might deteriorate the result? Assuming that correct dependencies are useful, it is therefore helpful to know how confident the parser is about its decisions, so that only reliable results are used.

I have tried to estimate such confidence values with help of probabilities that I could compute with parser's model. Similarly to the idea about difficult and easy classification cases I have thought that easy cases should be those which are also more likely to be correct. Unfortunately, I have found out that the parser's decisions are sometimes erroneous even though it had a 100% confidence according to its model. Even though the tendency that easier instances were less error-prone was there, the overall result was not satisfactory.

A similar idea of using the probabilities computed with parser's model was later reported in [84], a work about non-deterministic parsing with backtracking. In this paper the author uses these probabilities not only to select the most probable, but also the second-most probable decision, which is used in case of backtracking. However, this strategy is considered to be only a baseline in non-deterministic parsing. For a meaningful improvement additional adjustments have to be done.

Lesson learned:

In order to validate the quality of results it is not possible to use the same model which was used to produce them. Despite the fact that, similarly to feature usefulness, I thought that it is one of the top priorities for parser to be able to assess its results, so that the follow-up applications are warned about uncertain results, there is no system able to do that up to present day.

4.2 Achievements

Concluding this part of the thesis I will present all achievements in dependency parsing which arose during my work. Most of these succesful experiments have been published or are yet to appear on various international conferences.

Efficient Feature Extraction I was able to show that the traditional belief that the efficiency of a dependency parser is mainly dependent on the complexity of the parsing algorithm, and thus the number of transitions the parser requires to construct the dependency analysis, is wrong. I have used Java profiling technology in order to analyse how much execution time is spent in each stage of processing and found out that the overwhelming amount of time is spent on feature extraction. Since the bigger number of states does not automatically mean that features have to be extracted more often as well, since e.g. non-permissible states can be treated differently from the normal states, one has to actually analyse how many feature extractions are necessary to construct the dependency analysis. Some algorithms are more suitable for feature extraction than the others and in the end that is the main criterion for the efficiency. I was able to demonstrate that the Covington's strategy with quadratic complexity, which I found most suitable for efficient processing, greatly outperforms the usually preferred Nivre's algorithms with linear complexity, which are not suitable for efficient feature extraction. I have published these findings as a paper, which is to appear at KI-2012.

I have already discussed the differences between Covington's and Nivre's parsing strategies and the reason why the first one has quadratic and the latter have linear complexity, namely because Covington's strategy does not eliminate fully processed tokens from the agenda. Therefore Covington's strategy requires more configurations in order to process a sentence than Nivre's algorithms do. However, the configurations of Covington's algorithm are of two different types: permissible and non-permissible. Non-permissible configurations are those pairs

of words which violate well-formedness constraints and for which there is no need to construct feature vectors and one can automatically choose the NO-ARC operation. Permissible operations allow at least one of left-arc or right-arc operations and therefore require feature extraction in order to predict the correct transition. Both configuration types require a permissibility check, however, as my profiling findings have shown it is a very cheap procedure. The expensive feature extraction is only necessary for permissible configurations. In practice the amount of permissible configurations is approximately the same as with Nivre's algorithms, e.g. for English test data with 1337 sentences 63916 such configurations are required with Covington's algorithm, 64137 with arc-eager and 65148 with arc-standard. Of course with Covington's algorithm one has a lot of other configurations which also require computation in order to determine that they are non-permissible, but as I have already mentioned, the most computationally expensive part of the processing accounts for feature extraction and we see that all algorithms have a comparable number of them, despite the theoretical differences in complexity.

Now let us analyse why Covington's strategy is much more appealing despite the similar amount of feature extraction steps required for all algorithms. I am going to exemplify this by analysing the processing of the following English sentence: $Economic_1$ $news_2$ had_3 $little_4$ $effect_5$ on_6 $financial_7$ $markets_{8 \cdot 9}$ with the following dependency structure: (1,NMOD,2), (2,SBJ,3), (3,ROOT,0), (4,NMOD,5), (5,OBJ,3), (6, NMOD, 5), (7, NMOD,8), (8, PMOD,6), (9,P,3) (see Section 2.1.1. for dependency relation notation). I have used MaltParser, which has all algorithms implemented, with option "-m testdata" in order to analyse how many transitions are necessary to parse the data. In following i will list all transitions to examplify how exactly the parses look like. Every line corresponds to a configuration, consisting out of the predicted transition type, e.g. SH=shift, RE=reduce, LA=left-arc or RA=right-arc, plus the predicted type of the depenedency relation (applicable only for LA and RA of course), as well as the contents of both input and buffer stacks (cf. [62] for more details about transitions and notation). E.g. with Nivre's arc-eager strategy it took 16 transitions to parse the example sentence:
SH([0], [1..9]),
LA~NMOD([0,1], [2..9]),
SH([0], [2..9]),
LA~SBJ([0,2], [3..9]),
RA~ROOT([0], [3..9]),

SH([0,3], [4..9]),
LA~NMOD([0,3,4], [5..9]),
RA~OBJ([3], [5..9]),
RA~NMOD([0,3,4,6], [7..9]),
SH([0,3,4,6,7], [8,9]),
LA~NMOD([0,3,4,6], [8,9]),
RA~PMOD([0,3,4,6,8], [9]),
RE([0,3,4,6], [9]),
RE([0,3,4], [9]),
RE([0,3] ,[9]),
RA~P([0,3,9], []).

With Nivre's arc-standard 17 transitions were necessary:
SH[0], [1..9],
LA~NMOD[0,1], [2..9],
SH([0], [2..9]),
LA~SBJ([0,2], [3..9]),
SH([0], [3..9]),
SH([0,3], [4..9]),
LA~NMOD([0,3,4], [5..9]),
SH([0,3], [5..9]),
SH([0,3,5], [6..9]),
SH([0,3,5,6], [7..9]),
LA~NMOD([0,3,5,6,7], [8,9]),
RA~PMOD([0,3,5,6], [8,9]),
RA~NMOD([0,3,5], [6,9]),
RA~OBJ([0,3], [5,9]),
SH([0], [3,9]),
RA~P([0], [3]),
RA~ROOT([0], []).

Covington's algorithm is not stack-based, but uses the indexes j and i (included in the parantheses), in order to examine a pair of words and determine whether there should be a dependency relation between them (RA and LA) or whether one should proceed to the next pair (SH=shift, which increments j by 1 or NA=no-arc, which decrements i by 1). For italic typed configurations neihter LA or RA transitions are allowed, because they violate permissibility constraints. Again, the type of the dependency relations is also provided for RA and LA transitions.

For Covington's algorithm there are 33 word pairs examined, however, 17 of them are not permissible and thus there are only 16 permissible transitions:
SH(0,1),
LA~NMOD(1,2),
SH(0,2),
LA~SBJ(2,3),
(1,3),
RA~ROOT(0,3),
SH(3,4),
LA~NMOD(4,5),
RA~OBJ(3,5),
(2,5),
(1,5),
(0,5),
RA~NMOD(5,6),
(4,6),
(3,6),
(2,6),
(1,6),
(0,6),
SH(6,7),
LA~NMOD(7,8),
RA~PMOD(6,8),
(5,8),
(4,8),
(3,8),
(2,8),
(1,8),
(0,8),
NA(8,9),
NA(7,9),
NA(6,9),
(5,9),
(4,9),
RA~P(3,9).

In order to predict what transition should be performed in which parser state, the parser state is transformed into a feature vector and according to

the previously learned model the best transition is selected. The algorithms presented in this paper require a similar number of feature templates in order to achieve similarly competitive performance. In MaltParser arc-standard default algorithm runs with 21 different templates, arc-eager with 22 and Covington's algorithm also uses 22 feature templates. The templates use address functions and attribute functions[61] in order to extract features relevant for the state. E.g. given the state (3,5) for Covington's algorithm the function POSTAG (Right[0]) returns the POS tag of the word with index 5. Similarly given some buffer stack for Nivre's algorithms the function FORM(Input[2]) returns the word form of third-top token on the buffer stack.

During the processing many features are used more than once. E.g. POSTAG (Right[0]) is the same for configurations (3,9), (6,9), (7,9) and (8,9). Similarly for Nivre's algorithm POSTAG (Stack[0]) is the same when both (4,NMOD,5) and (6,NMOD,5) dependencies are constructed. Thus the POS value of the tokens 9 and 5, respectively, can be looked up only once and the index of this feature can be stored in memory as long as sentence is processed. This way a lot of time can be saved.

The decisive difference between the algorithms is that many other features can only be reused in Covington's and not in Nivre's algorithms. E.g. POSTAG (Left[0]) or POSTAG (Left[1]) is the same for configurations (3,4), (3,5) and (3,9), as well as POSTAG (Right[0]), POSTAG (Right[1]), POSTAG (Right[2]) etc. is the same for configurations (1,2) and (0,2). One can basically precompute all values since if Right[0] = j, then Right[1] is definitely $j+1$, Right[2] = $j+2$, Right[3] = $j+3$. For Nivre's algorithms the reusability of features is limited, because one never knows how the stacks will look like. If Stack[0] is j, then Stack[1] might be $j-1$, $j-2$, $j-3$ or any other token and it would be too memory intensive to keep all possibilities in memory until it is clear which one of them is correct. The reusability of static features considerably improves the performance of an algorithm, since it is no longer necessary to look up the value of a feature and then its index in a global alphabet so often. Instead, we consult the global mapping only once for all features which are used many times and store them in a different local (i.e. valid only within the current sentence) data structure from where they can be retrieved much faster. E.g. if we are in the configuration (2,3), then Right[0] = 3 and we know that Right[1] = 4, Right[2] = 5 and Right[3] = 6, thus POSTAG (Right[0]) = VBD, POSTAG (Right[1]) = JJ, POSTAG (Right[2]) = NN and POSTAG (Right[3]) = IN. We look up their indexes in the global alphabet and store them. When we are in the configuration

(0,3) we know that all these features are exactly the same as in (2,3) and we can reuse the already looked up indexes.

Additionally, in order to compensate for the lack of a kernel, which creates conjoined features implicitly, one has to add artificial feature combinations manually. In MaltParser's feature models for LibLinear around 40% are feature combinations, which are concatenations of basic features. E.g. merge2(POSTAG (Right[0]), POSTAG (Right[1]) for configuration (2,3) would return the value VBD~JJ and merge3((POSTAG (Right[0]), POSTAG (Right[1], POSTAG (Right[2])) - VBD~JJ~NN.

We have seen that for parsing English development data one needs around 65000 configurations. If there are 40% feature combinations, which require 1-2 concatenations each, that are around 12 string concatenations per configuration and 780,000 overall. Since for security reasons String is an immutable basic type in Java, each time you append something a new String is created, the old value is stored the new value is added, and the old String is thrown away. The longer the strings the longer the concatenations take, but even for typical features length of ~10 characters a concatenation takes around 0.25-0.3 microseconds. For 780,000 feature combinations it would mean around 0.2 seconds, i.e. around 20% of the whole execution time if one wants to parse a sentence in less than 1ms.

Therefore it is even more important that feature combinations are reused whenever possible, because they contain additional costly string operations. So instead of using feature combinations like merge2(POSTAG (Right[0]), POSTAG (Left[0])) and merge2(POSTAG (Right[1]), POSTAG (Left[1])), which potentially have different values in any configuration and thus can be used only once, it is better to use combinations with reusable values, e.g. merge2(POSTAG (Right[0]), POSTAG (Right[1])) and merge2(POSTAG (Left[0]), POSTAG (Left[1])). This way the values returned by these templates can be reused in configurations (i,j), which share the same j or i, respectively.

It is important to note that even though String operations are expensive in Java there are no alternatives. Tricks like translating features to integers and substituting concatenation by multiplication do not work better, since they require an additional mapping from the original String values to ints and the look up in such large collections is even more expensive than concatenation. Assuming a HashMap size of one million (not an unusual number of features for large treebanks) the execution of a get() method (required to look up the int value for a certain feature string) takes more than 200 times longer than

a concatenation. Even for smaller alphabets, e.g. with 100000 values, it takes 30-40 times longer.

My implementation of MDParser tries to reuse static features whenever possible. Additionally, I have modified the feature models, so that they contain as many reusable feature combinations as possible. Despite the worse theoretical complexity, I have shown that in practice the worst case never occurs and other properties of an algorithm can be much more important. In particular the suitability of an algorithm for efficient feature extraction is decisive, since most of the execution time required for processing is spent on this subtask.

Error Identification and Correction Every treebank contains annotation errors, which are harmful for data-driven parsers, which learn to replicate them. Despite the fact that the quality of annotated resources is of essential importance for the quality of resources derived from them, there is not much literature available on this topic. A notable exception is the group around Detmar Meurers and some publications about their approach for finding errors called *variation detection*[30]. I propose a different method, which is complementary to variation detection and is able to find additional errors. Moreover, it is not only able to recognise inconsistencies, but also proposes a correction. The proposed method was published in the proceedings of ACL 2011[82].

The construction of an annotated corpus involves a lot of work performed by large groups. However, despite the fact that a lot of human post-editing and automatic quality assurance is done, errors can not be avoided completely[31]. The quality control is usually performed manually by developers, often by receiving feedback from the community working with the resources. The only work, that we were able to find, which involved automatic quality control, was done by the already mentioned group around Detmar Meurers. This work includes numerous publications concerning finding errors in phrase structures[31] as well as in dependency treebanks[13].

The idea behind *variation detection* is to find strings, which occur multiple times in the corpus, but which have varying annotations. This can obviously have only two reasons: either the strings are ambiguous and can have different structures, depending on the meaning, or the annotation is erroneous in at least one of the cases. The idea can be adapted to dependency structures as well, by analysing the possible dependency relations between same words. Again different dependencies can be either the result of ambiguity or errors. If the difference between the different readings is due to the ambiguity, i.e. they are

not errors, then they usually occur many times in the corpus. On the other hand, if one of the readings is erroneous, then it usually occurs only very few times. According to these heuristics the method can select error candidates which are then manually evaluated.

I propose a different approach. The idea is to take a dependency treebank and to train models with different parsers, e.g. I took the state-of-the art graph-based MSTParser and the transition-based MaltParser. Then parse the data, which was used for training, with both parsers. The idea behind this step is that one basically tries to reproduce the gold standard, since parsing the data seen during the training is very easy (a similar idea in the area of POS tagging is very broadly described in [79]). Indeed both parsers achieve accuracies between 98% and 99% UAS. The reason why the parsers are not able to achieve 100% is on the one hand the fact that some of the phenomena are too rare and are not captured by their models. On the other hand, in many other cases parsers do make correct predictions, but the gold standard they are evaluated against is wrong. I have investigated the latter case, namely when both parsers predict dependencies different from the gold standard (I did not consider the correctness of the dependency label). Since MSTParser and MaltParser are based on completely different parsing approaches they also tend to make different mistakes[67]. Additionally, considering the accuracies of 98-99% the chance that both parsers, which have different foundations, make an erroneous decision simultaneously is very small and therefore these cases are the most likely candidates when looking for errors.

Furthermore, I did not perform a manual evaluation of error candidates, but tried to automatise the procedure. Therefore I have used MDParser as the third parser. The exact procedure looks as follows:

1. Automatic detection of error candidates, i.e. cases where two parsers deliver results different to gold-standard.
2. Substitution of the annotation of the error candidates by the annotation proposed by one of the parsers (in our case MSTParser).
3. Parse of the modified corpus with a third parser (MDParser).
4. Evaluation of the results.
5. The modifications are only kept for those cases when the modified annotation is identical with the one predicted by the third parser and undone in other cases.

For the English dependency treebank I have identified 6743 error candidates, which is about 0.7% of all tokens in the corpus. During this experiment I have found out that the result of MDParser significantly improves: it is able to correctly recogonise 3535 more dependencies than before the substitution of the gold standard. 2077 annotations remain wrong independently of the changes in the gold standard. 1131 of the relations become wrong with the changed gold standard, whereas they were correct with the old unchanged version. The changes to the gold standard when the wrong cases remained wrong and when the correct cases became wrong are undone. I suggest that the 3535 dependencies which became correct after the change in gold standard are errors, since a) two state of the art parsers deliver a result which differs from the gold standard and b) a third parser confirms that by delivering exactly the same result as the proposed change. However, the exact precision of the approach can probably be computed only by manual investigation of all corrected dependencies.

So far I have tried to evaluate the precision of the approach for the identified error candidates. However, it remains unclear how many of the errors are found and how many errors can be still expected in the corpus. Therefore I have also attempted to evaluate the recall of the proposed method. In order to estimate the percentage of errors, which can be found with this method, I have designed the following experiment. I have taken sentences of different lengths from the corpus and provided them with a "gold standard" annotation which was completely (=100%) erroneous. I have achieved that by substituting the original annotation by the annotation of a different sentence of the same length from the corpus, which did not contain dependency edges which would overlap with the original annotation. E.g consider the following sentence in the (slightly simplified) CoNLL format:

1 Not RB 6 SBJ
2 all PDT 1 NMOD
3 those DT 1 NMOD
4 who WP 5 SBJ
5 wrote VBD 1 NMOD
6 oppose VBP 0 ROOT
7 the DT 8 NMOD
8 changes NNS 6 OBJ
9 . . 6 P

I substitute its annotation by an annotation chosen from a different sentence of the same length:

1 Not RB 3 SBJ
2 all PDT 3 NMOD
3 those DT 0 NMOD
4 who WP 3 SBJ
5 wrote VBD 4 NMOD
6 oppose VBP 5 ROOT
7 the DT 6 NMOD
8 changes NNS 7 OBJ
9 . . 3 P

This way we know that a well-formed dependency tree was introduced (since its annotation belonged to a different tree before) to the corpus and the exact number of errors (since randomly correct dependencies are impossible). In case of this example 9 errors are introduced to the corpus. In the experiment I have introduced sentences of different lengths with overall 1350 tokens. I have then retrained the models for MSTParser and MaltParser and have applied the methodology to the data with these errors. I have then counted how many of these 1350 errors could be found. The result is that 619 tokens (45.9%) were different from the erroneous gold-standard. That means that despite the fact that the training data contained some incorrectly annotated tokens, the parsers were able to annotate them differently. Therefore I suggest that the recall of our method is close to the value of 0.459. However, of course it is unknown whether the randomly introduced errors in this experiment are similar to those which occur in real treebanks.

The interesting question which naturally arises at this point is whether the errors found with this method are the same as those found by the method of variation detection. Therefore I have performed the following experiment: I have counted the numbers of occurrences for the dependencies $B \rightarrow A$ (the word B is the head of the word A) and $C \rightarrow A$ (the word C is the head of the word A), where $B \rightarrow A$ is the dependency proposed by the parsers and $C \rightarrow A$ is the dependency proposed by the gold standard. In order for variation detection to be applicable the frequency counts for both relations must be available and the counts for the dependency proposed by the parsers should ideally greatly outweigh the frequency of the gold standard, which would be a great indication of an error. For the 3535 dependencies that are classified as errors the variation detection method works only 934 times (39.5%). These are the cases when the gold standard is obviously wrong and occurs only few times, most often - once, whereas the parsers propose much more frequent dependencies. In all

other cases the counts suggest that the variation detection would not work, since both dependencies have frequent counts or the correct dependency is even outweighed by the incorrect one.

Finally, I will provide some of the example errors, which I was able to find with this approach. Therefore I will provide the sentence strings and briefly compare the gold standard dependency annotation of a certain dependency within these sentences.

Together, the two stocks wreaked havoc among takeover stock traders, and caused a 7.3% drop in the DOW Jones Transportation Average, second in size only to ***the stock-market crash*** *of Oct. 19 1987.*

In this sentence the gold standard suggests the dependency relation *market* $\rightarrow$ *the*, whereas the parsers correctly recognise the dependency *crash* $\rightarrow$ *the*. Both dependencies have very high counts and therefore the variation detection would not work well in this scenario.

Actually, it was down only a few ***points at*** *the time.*

In this sentence the gold standard suggests *points* $\rightarrow$ *at*, whereas the parsers predict *was* $\rightarrow$ *at*. The gold standard suggestion occurs only once whereas the temporal dependency occurs 11 times in the corpus. This is an example of an error which could be found with the variation detection as well.

Last October, Mr. Paul paid out $12 million of CenTrust's cash – plus ***a $****1.2 million* ***commission*** *– for "Portrait of a Man as Mars".*

In this sentence the gold standard suggests the dependency relation \$ $\rightarrow$ *a* , whereas the parsers correctly recognise the dependency *commission* $\rightarrow$ *a*. The interesting fact is that the relation \$ $\rightarrow$ *a* is actually much more frequent than *commission* $\rightarrow$ *a*, e.g. as in the sentence *he caught up an additional $1 billion or so.* (\$ $\rightarrow$ *an*). So the variation detection alone would not suffice in this case.

The results show that both approaches are rather complementary and find different types of errors. However, I have evaluated my approach automatically, whereas variation detection was always evaluated manually. Additionally, I have tried to estimate the overall number of errors, which seems to be over 1% of the total size of a corpus, which is expected to be of a very high quality. A fact that one has to be aware of when working with annotated resources and which is one of the important findings of my work.

Experiments with sentence usefulness Treebanks for some languages are very big and learning curve experiments show that almost the same result can already be achieved with less amount of data. For other languages the treebanks

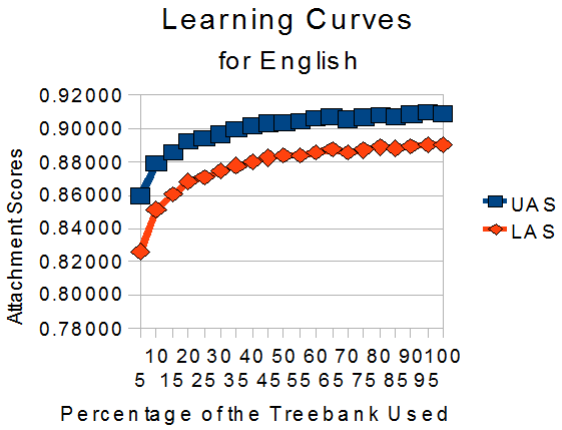

Figure 3: Learning Curves for English

are too small and much better results could be achieved if more annotated data were available. Since annotation of corpora is an expensive process, it is important to control it as good as possible. A very interesting question here is what type of data exactly is useful and should be annotated first, because it is both more beneficial for the performance and requires less effort to be annotated. In following I will describe a method which allows to measure the usefulness of the material one wants to annotate and thus helps to control the process. This approach has been published and is to appear in KONVENS 2012.

In order to measure whether a sufficient amount of annotated data is available for a certain application it is best to look at its learning curve with increasing size of training data. The following two diagrams show how dependency parsing performance for English and Finnish[37] treebanks[5] looks like depending on the percentage of the available data used.

Both English and Finnish learning curves are very steep in the beginning when little data is available. However, for English the performance hardly changes already when 50-60% of data is used, whereas for Finnish even at 100% the gradient of the curve is still quite big (one interval on the y-axis for Finnish is 0.10 AS on the contrary to 0.02 for English) and thus the performance would in all probability profit from additional annotated data.

For constructing these two diagrams forty models had to be trained and therefore the parameters used were optimized to speed rather than accuracy

[5]I am particularly grateful to Filip Ginter and Katri Haverinen for providing me with different versions of the Finnish treebank

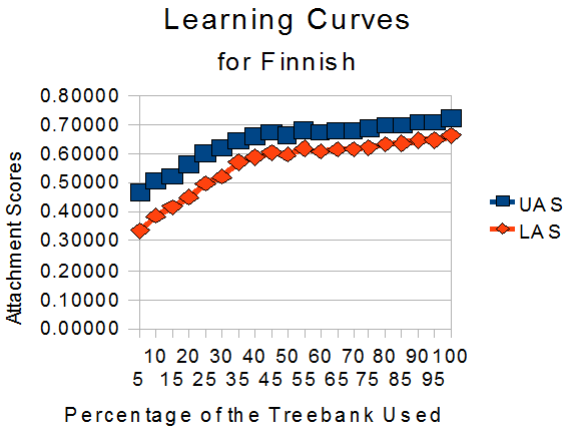

Figure 4: Learning Curves for Finnish

(e.g. linear classification was used). Therefore the above performance does not represent the state of the art accuracies for these two languages and serves only the purpose of visualising the necessity of expanding annotated resources.

I have investigated whether it is possible to measure the usefulness of annotating and adding a sentence to an existing treebank and whether this value significantly differs across different sentences. I have ascertained that it is indeed possible and show that a lot of time can be saved and much better results can be achieved, if one a) does not waste time on annotating bad sentences and b) concentrates on annotating good sentences. Finally, I have constructed a statistical model able to discriminate between those two types and evaluated it.

Since I do not have any knowledge of Finnish language and was not able to annotate sentences myself, I have left out a portion of annotated sentences out of treebank and added them in the process as if they were newly annotated by me.

A similar work exists in the area of active learning[56]. The work has exactly the same motivation of selecting only that material for annotation, which is beneficial for the performance of one's system and not wasting the time on the rest. The authors even propose not to annotate the whole sentence manually, but rather process the sentence with a parser and only manually overwrite those dependencies which are actually useful. The method of selecting good material proposed by Mirroshandel and Nasr is different from what I have investigated. Their idea is that those structures which are most error-prune require annota-

tion, since obviously the model is missing the knowledge to process them. They then manually annotate the wrong parts, relearn the model and hope that the parser has learned to deal with the structures which went wrong before. Thus the most useful sentence is the one which contained the most mistakes prior to its manual annotation. On the other hand sentences which are already parsed correctly are not useful and do not require annotation. My proposal is that a sentence is useful and requires annotation if its inclusion to the training data improves the performance of the parser on the test data. This way the most useful sentence is the one whose annotation has the biggest positive effect on the performance of the parser on the test data.

Idefine usefulness (su) of a sentence (s_i) as the influence of a sentence on the accuracy (acc) of a system for the test data (TE_m) when added to the training data (TR_n) of that system:

$su(s_i) = acc(TR_n + s_i; TE_m) - acc(TR_n; TE_m)$

For accuracy I have used the labeled attachment score. In order to compute a good estimate for sentence usefulness I have used several training data sets and several test data sets and averaged the change in accuracies across all experiments. For experiments described here I have taken the Finnish treebank (6375 sentences) and split it into 2 training data sets (30% of the whole data each), 2 test data sets (10% each) and left the remaining 20% (1276 sentences) for experimenting. Thus for each of these 1276 sentences I have computed the accuracy for all (4) combinations and averaged over their number:

$su(si) = (acc(TR1{+}si; TE1){-}acc(TR1; TE1){+}acc(TR1{+}si; TE2){-}acc(TR1; TE2){+}$
$acc(TR2 + si; TE1){-}acc(TRn; TE1) + acc(TR2 + si; TE2){-}acc(TR2; TE2))/4$

In order to test whether the sentence usefulness measure serves its purpose, namely as a selection criterion for sentences which should be added to the training data foremost, I have performed several experiments. I have computed accuracies for randomly selected n sentences and compared it to the accuracies when the best n sentences where selected according to my metric. E.g. when 20 sentences were randomly added to the training data the accuracy improved by 0.08% LAS on average, whereas with 20 best sentences the improvement was 0.3% LAS. For 50 random sentences the improvement was 0.13% LAS, whereas for the best 50 it was 0.48% LAS.

Having shown that different sentences have different impact on the accuracy and that it is beneficial to add good ones first, I have investigated whether it is possible to automatically predict this value using a statistical classifier. Basically, we are not interested in the exact sentence usefulness decimal value,

but we rather want to know whether a sentence is worth annotating or not. Therefore we can group sentences into some discrete usefulness classes. The core problem is binary, since we want to discriminate between good sentences we want to annotate and the rest. However, the number of good sentences is small and the rest is big, which makes the problem very imbalanced and the prediction difficult. Therefore I have experimented with several different divisions and could achieve the best results when the data is split into four classes of equal number: 1 being the worst, 2 being slightly better, 3 – good, 4 – best. Even though that we are still interested only in the class 4, this partitioning allows us to better optimise the performance of the classifier. In cases when the decision is not very certain, usually the confusion is only between two classes[41]. E.g. when two good classes (3 and 4) compete, a mistake is not that bad as e.g. in the case of a good and a bad class (1 and 4), where the risk of making a severe mistake is much higher. In the simple binary case one would not be able to differentiate between different types of mistakes that well.

First, I have tried to represent each sentence as a feature vector and then learn the corresponding classes using linear SVMs. However, either because I was not able to come up with enough good sentence-level features or because the size of the training data was not big enough (I have used 20% of the data left for experimenting, which then again was randomly split into 90% for training and 10% for testing), the results were not satisfactory. I have then decided to do the classification on the word level: for every word in a sentence of class c, I have constructed a feature vector and assigned it the class c in the training phase. E.g. for a class 3 sentence of length 10 I have constructed 10 feature vectors, one for each word, of the class 3. The feature templates which I have used were:

sentence-level features: length of the sentence, number of punctuation tokens, number of verbs, number of pronouns, number of conjunctions, number of adverbial modifiers

word-level features: word form of the current word, word form of the next word, POS of the current word, POS of the next word, word length, dependency type, word novelty

These particular features were selected after some semi-automatic feature engineering, during which I have tried all kinds of different POS tags and dependency types as features.

In order to detect punctuation, verbs and pronouns I have simply used the Finnish POS tags: PUNCT, V and PRON respectively. For adverbial modifiers

Algorithmus 4 Voting procedure for usefulness classes

```
if 4 ∈ V
  weight(1) = count(1)*2 in V
  weight(2) = count(2) in V
  weight(3) = count(3) in V
  weight(4) = count(4) in V
  return argmaxi(weight(i))
else
  return majority_voting(V)
```

and dependency types in general I have parsed every sentence with MaltParser and used the label predicted by the parser (not the gold standard that also was available). The novelty of a word is a binary feature: for the given word form we look whether it already ever occurred in the training and test data or whether it is new. In the test phase we would then predict a class for every word in a sentence and perform a voting procedure in order to infer the sentence usefulness class for the whole sentence. Given a set of votes , where , the voting procedure looks like that:

So basically if V does not contain words classified as class 4, I have simply performed majority voting for the remaining classes. Otherwise all votes for other classes are counted, whereas the votes for the worst class are always counted twice in order to avoid a severe mistake whenever there is a chance that a sentence might belong to class 1.

The evaluation of the model was not straightforward. The first metric which I have tried out was accuracy: the percentage of correctly classified classes. However, since we are not equally interested in all classes, but are rather interested in correctly finding the class 4, the metric was not very helpful.

That is why the second attempt was to compute precision and recall only for the class 4. However, different mistakes have different impact on the result. A confusion between class 1 and class 4 is different from a confusion between classes 3 and 4. Precision and recall metrics are thus not helpful neither.

The next step was to compute a confusion matrix for all classes and try to minimise the number of actual instances of 1 and 2 predicted as 4 on the one hand and maximising the number of actual 4 predicted as 4 on the other hand.

In a real-world scenario the number of potential sentences which can be annotated is infinite and therefore the recall of the method might be arbitrary low and it will still find enough good candidates to annotate. It is merely important that the precision is high, so that the sentences which are annotated

are really beneficial for the task. However, in our experiment the amount of data was small: in 20% of the sentences left for testing, i.e. 250 sentences, 57 belonged to class 4. Therefore I could not decrease recall because otherwise not enough sentences would have been found in order to clearly demonstrate that the approach outperforms the baseline of selecting sentences randomly.

I have run 10 tests randomly selecting 80% sentences for training and 20% for testing and averaged the performance on the confusion matrix:

In around 16% of all cases actual 1s were classified as 4s. In around 26% of all cases actual 2s were classified as 4s, in around 8% of all cases actual 3s were classified as 4s and in around 50% of all cases actual 4s were correctly classified as 4s.

Thus this method allows to select good sentences significantly more often (namely twice as often) than when doing it randomly. Again, if there were more data available for testing, it would be possible to tune the precision at costs of recall, improving the performance even further. The experiment with gold-standard usefulness classes, suggests that it is possible to gain accuracy up to four times faster when annotating sentences proposed by this method compared to random selection.

I suppose that the sentence usefulness is not a constant parameter, but rather changes with the growing size of the training data. Because of the limited size of data for experimenting I could not investigate whether this metric is still better than random when hundreds or thousands of sentences are added to the original corpus and/or whether the model has to be adapted and retrained on the new data, because in the course of annotation different units might become useful than it was the case prior to the treebank extension. Most probably the latter case is true and the model has to be regularly adopted to the changes in the treebank, however, I do not know whether it will be an easier or a harder task to predict new good candidates for annotation. On the one hand with the growing size it should become more difficult to find new beneficial sentences for the treebank, but on the other hand with more data available a more accurate model for prediction can be constructed.

Another interesting point is to analyse what actually makes some sentences better than other. It was important to make sure that longer sentences are not always automatically better, simply because they have more tokens and thus provide more information. So in one of the experiments I have sorted all sentences according to their usefulness and looked at their length. The result clearly shows that even though the best sentences are on average slightly longer

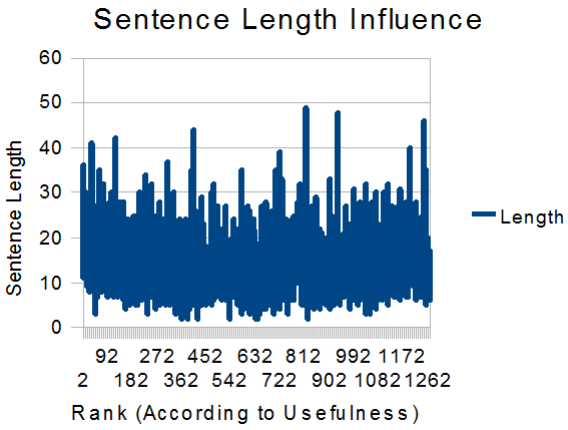

Figure 5: Sentence Length Influence

there is still a huge amount of long sentences which were not useful. In principle the approach might be further optimised by not only finding most promising but also short sentences, which do not require much time to annotate. The figure 5 demonstrates that there are lots of short sentences which are among the most useful ones and long sentences which are among the most useless one and that the length of a sentenes is actually quite an irrelevant feature. Since the time effort depends on the number of tokens one has to annotate, that is an important finding, because it means that not only one is able to find good sentences but also those which can be annotated with less effort.

Unfortunately, I am not a speaker of Finnish and I was not able to analyse neither the content of the most useful sentences nor their properties. It would be interesting to find out whether these sentences have certain linguistic structures or contain specific lexical entries. Neither could I investigate whether the useful sentences we are able to detect overlap with those predicted by the approach proposed by the already mentioned approach by Mirroshandel and Nasr. The approaches are very different: they work in a bottom-up fashion by optimising the model for the sentences with poorest performance until the model performs well for most sentences and we work in a top-down fashion by optimising the model for sentences which improve the performance for as many other sentences as possible in the test data. Intuitively, the first and the latter sentence types are not the same and thus it would also be interesting to combine both approaches.

This work has shown the necessity of extending annotated resources on the

example of Finnish. Since the annotation process is expensive and tedious I have proposed an approach for selecting only the most promising sentences for annotation. I have explained a sentence usefulness metric and how it can be predicted for new sentences. The experiments show that it is possible to select good sentences significantly better than simply annotating random sentences. If gold standard values for sentence significance are used, the accuracy of the parser increases four times faster than when random sentences are added to the training data. With the values predicted by my model it is still twice as fast. I have also discussed some potential problems with the metric if the size of the initial treebank changes considerably. However, on the other hand it might become easier to accurately predict good sentences if more data becomes available.

Investigation of differences between linear and non-linear kernel-based learners For some applications the quality of results is the main priority, whereas for some others there are also requirements to the speed of the processing. Accuracy and efficiency are thus two main properties of dependency parsers. A couple of years ago accurate parsing was slow, and fast parsing was relatively inaccurate. Accurate parsers like MaltParser or MST Parser required many hours or even days to train a model from a large treebank, e.g. for English. Applying the parsers was also a slow process, since the speed was dependent on the number of training instances. A usual speed of such older parsers like MaltParser (using LibSVM), MST Parser or Stanford Parser is about 2-4 sentences per second. With faster machine learning methods, e.g. TiMBL[26] for MaltParser, the accuracy was 3-4% (depending on the language) lower, but the training and application of the system was much less time-intensive. The present day linear classifiers with good feature models, which contain artificial feature combinations, are even only around 1% behind the complex kernel-based methods. At the same time they are much more faster and are able to parse hundreds of sentences per second. MaltParser, using linear classification approach (using LibLinear library is able to parse about 9000 tokens per second, which corresponds to around 400 English sentences of average length (24.5 tokens)).

The following table sums up the performance of MaltParser with two different machine learning libraries:

	LibSVM	LibLinear
Accuracy (UAS/LAS)	92.1 / 90.4	90.9 / 89.3
Training Time	6h21m51s	5min23s
Parsing Time	371s	3.69s

Table 1: MaltParser(LibLinear) vs MaltParser(LibSVM)

The natural question, which arises at this point is whether a gain of 1.1% accuracy justifies 70 times longer training times and 100 times longer parsing times.

I have performed a detailed analysis, which has shown that 86.9% (29015 tokens) of all dependencies are equally recognised by both models and are correct. 2.9% (968 tokens) are correctly recognised only by LibSVM model and 1.7% (577 tokens) – only by LibLinear, the rest is not recognised by neither of both. It is especially interesting whether the accuracy gain of LibSVM model justifies the hundred-fold decrease in efficiency and therefore I have performed a detailed analysis of the 2.9% of dependencies which are only recognised when using LibSVM. The analysis consisted out of the following steps:

1. Investigation whether these 2.9% are just an ordinary extract of the data, which contains approximately the same dependencies as any other sample section of the data.

2. Investigation whether LibSVM performs better simply due to overfitting the data.

3. Investigation whether the quadratic kernel of LibSVM, being a more complex approach, helps to recognise more difficult dependencies better than the simpler linear SVMs.

Therefore I have done the following three experiments:

1. I have computed the distribution for different dependency types for the whole data and for the 968 tokens, which are only correctly processed by LibSVM, and compared them. Most dependency types indeed have approximately the same distribution as the entire data. These dependency types account for 61.8% of the 968 tokens. However, the rest is quite different and occurs either much more or much less often, i.e. at least by a factor of 2, than according to the distribution of the whole data. E.g. the rather unimportant dependency type P (punctuation) accounts

for 11.02% of all dependencies in the English data. In these 968 tokens punctuation occurs 253 times, i.e. 26.13% of all dependencies. Among other dependencies which occur with an above-average frequency were tokens of type COORD (coordination), which occurred 65 times (6.71%), whereas the average is 2.65%, and APPO (apposition) - 32 times (3.30%), average 1.41%. The above-average of correctly recognised coordinations, punctuations and appositions gives rise to the suspicion that LibSVM is particularly suitable for longer sentences, with relative clauses and appositions, as well as punctuation associated with these phenomena. Indeed, our experiments showed that whereas the average sentence length of the sentences in the data is 25.01 words, the average length of the sentences, which are correctly parsed by LibSVM but not LibLinear is 30.67, i.e. much longer sentences.

2. Any treebank contains a certain amount of inconsistency due to the nature of its creation. I have already shown in the *Error Identification and Correction* section that around 1% of all annotations in the English treebank is erroneous. The idea was to test whether the number of inconsistencies is higher for the 2.9% of data we are investigating. After applying the approach to this portion of the corpus I was able to identify 86 errors (8.3%), which exceeds the expected average by several times. That means that actually a big portion of what is recognised as correct by LibSVM and not LibLinear is actually due to the erroneous gold standard. A similar experiment for LibLinear revealed that it only classified 41 tokens with erroneous gold standard. Thus LibLinear seems to be able to generalise better, whereas LibSVM merely replicates the gold standard, even if it is inconsistent.

3. Whereas many parsers achieve results around 90%, the score is merely an average over all dependencies. Among them are those which are rather easy and unimportant for applications, and those which are difficult and highly relevant for certain tasks. The performance of parsers for the latter dependencies is usually much lower. Examples for such dependencies are non-local dependencies[8] and unbounded dependencies[69] .I have taken the data with non-local dependencies and evaluated MaltParser (LibSVM) and MaltParser (LibLinear) on it in order to investigate whether there is some difference between the models for more difficult phenomena. The result has shown that there is absolutely no difference between both parsers

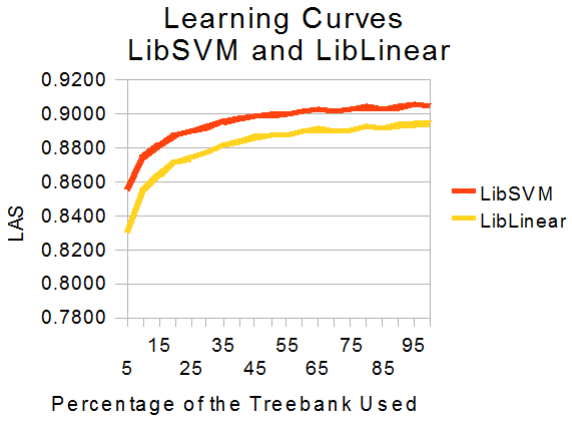

Figure 6: Learning Curves for LibSVM and LibLinear

and they were able to recognise exactly the same number of demanded annotations.

Apart from accuracy and efficiency machine learning methods might differ in another important property, namely the amount of required training data in order to perform well. Therefore we have also performed an experiment whether LibSVM and LibLinear versions of MaltParser require different size of training data in order to achieve high accuracy and thus would influence the choice of the machine learning library for languages with small dependency treebanks.

Similarly to the *Feature Usefulness* section I have investigated the learning curves for LibSVM and LibLinear for various instances of training data of different size. I have measured the performance of both LibSVM and LibLinear for each version of the training data used. English treebank is probably a bad example because for both learning approaches its size is more than enough. We see that the steepest improvement is only in the beginning and that with 40% of the data LibSVM's accuracy is just 1% lower than with the full corpus. With LibLinear it is very similar and it is also able to achieve an accuracy just 1% below its top performance with 40% of the data. Thus LibSVM does not provide any benefit for languages with much smaller treebanks, since LibLinear does not require a lot of training data neither.

The investigation shows that the higher score of LibSVM results from its better ability to replicate the gold standard. On the one hand one can therefore better parse longer sentences with more complex syntactic structure, like

appositions and coordinations, on the other hand LibSVM blindly replicates inconsistencies, which LibLinear does not because of a more general model. The experiment on ten non-local dependency types shows no difference between both models. The difference in efficiency is enormous, both training and application times differ by a factor of 70-100, which would make LibSVM not applicable for a wide range of application requiring fast processing (e.g. if processing is done online or the amount of data is big, for instance data from the web), whatever accuracy it might have had. Additionally, I have investigated whether the more complex LibSVM library performs better if less training is available. The experiments have shown that there is no difference in behaviour between LibSVM and LibLinear. I therefore think that parsing with a linear classifier is a much better option. It works much faster, trains a more general model which is less tuned to the data with its potential inconsistencies, does not require more data to train and achieves almost the same results, as our additional experiments with non-local dependencies have shown.

Part II

Evaluation

In the first part of my thesis I have presented the formalism of dependency grammar and described the properties of numerous parsers based on it. This diversity makes it difficult to decide which parser is the most suitable for one's needs.

For many NLP tasks, such as machine translation being the extreme example, the evaluation is at least as complex as the task itself. For dependency parsing for a long time it was not the case. In fact one of the most advocated advantages of dependency parsing was the facility of its evaluation. The easiness of dependency parsing evaluation consisted in that a dependency tree can be represented as a set of individual dependency relations and the percentage of correctly assigned relations could be thus computed in a very simple fashion. The proportion of correctly recognised head-modifier dependencies is called unlabeled attachment score (UAS). Since dependency relations are usually typed, the percentage of correctly recognised dependency relations including the type (e.g. subject relation) is called labeled attachment score (LAS).

Especially in the years 2007-2009, when the CoNLL-X shared tasks in dependency parsing[15] took place, the metric became predominant in the field. It allowed to compare and rank the results of different systems for the same data set. The analysis could be deepened to a comparison of performance for different dependency types, sentence lengths and further more fine-grained evaluation. I will refer to the CoNLL-X style evaluation as traditional evaluation, because it is also the way the phrase structure parsers were evaluated in the decades before (cf. [3]).

An alternative evaluation methodology which has become popular in the recent years is the application-oriented extrinsic evaluation. With this type of evaluation the parser result is not evaluated directly, but rather by measuring its contribution to solving some task for which it was applied. On the contrary to the traditional evaluation, this methodology does not involve comparing the ability to recognise some artificial categories and is very intuitive. I will refer to it as natural evaluation.

In this part of the thesis I am going to discuss the advantages and drawbacks of both evaluation methodologies, provide the state-of-the-art works and present my own work on the topic of dependency parser evaluation.

5 Traditional Evaluation

Automatic quantitative evaluation of results is of essential importance for any NLP task. The development of an NLP system is greatly facilitated if such evaluation is possible, since in this case the quality of improvements to the system can be measured in a cheap and fast way. This is of course also true for parsing. The most influential metric for parsing, PARSEVAL, was developed in the 1990s for PSG parsers. According to this metric the brackets delimiting constituents are first lined up. Afterwards the precision and recall of matching brackets are calculated. Precision is the percentage of correct brackets among all brackets predicted by the system and recall is the ratio of correct brackets in the result to the total number of brackets in the gold standard. A metric called F-score is usually used to combine the recall and precision values in one, by calculating their harmonic mean:

$F - score = \frac{precision * recall}{2 * precision * recall}$.

A modified version of PARSEVAL also required the brackets to be labeled correctly, i.e. not only the boundaries but also the phrase types had to be correct in order to satisfy the requirements for matching the brackets.

The advantages of the PARSEVAL metric are self-evident:

- It is fully automatic and does not require any manual human evaluation
- It is easy to compute and the calculation is fast
- It delivers a single score which is particularly suitable in order to compare different systems against one another

This metric also seemed to qualify for the needs of dependency parsing. In fact it is even easier, since there are no constituents in the dependency formalism and therefore instead of precision and recall the accuracy, i.e. simply the percentage of correct decisions, can be taken. Similarly to PARSEVAL, again both labeled and unlabeled accuracies, called labeled attachment score (LAS) and unlabeled attachment score (UAS) are possible:

$UAS = \frac{correct\,heads}{total\,words}$

$LAS = \frac{correct\,heads \wedge correct\,types}{total\,words}$

Additionally, the dependency-based evaluation also neutralised one of the most prominent disadvantages of PARSEVAL: favouring of minimal structure[18] and penalising of misattachments[48] more than once. The first is true because constituents of different lengths contribute equally to the score, whereas it is

obvious that shorter ones are less error-prone. The second is true because a single error can cause several crossing brackets or missing boundaries. This is not an issue for dependency parser evaluation, because dependency evaluation is word-based.

5.1 CoNLL Evaluation

In the years 2006-2008 three shared tasks on multilingual dependency parsing were organised within the scope of the Conference on Computational Natural Language Learning (CoNLL). In terms of this initiative treebanks for 21 languages were transformed into a uniform format, which allowed to evaluate one and the same parser over a variety of languages and a variety of parsers for one and the same data set. The performance on the test sections of the treebanks released during the shared tasks has become a de-facto standard when reporting results for dependency parsers in the literature. Additionally, the CoNLL format[15] turned out to be a particularly appealing representation form and also has become one of the standard representations.

The CoNLL parser evaluation embodied all the advantages of PARSEVAL evaluation and introduced the additional benefits of dependency-based evaluation. Up to the present day it is the most prominent and popular evaluation methodology and CoNLL scores are reported for any parser.

A comprehensive overview of the attachment scores for MSTParser and MaltParser can be found in [15]. However, the feature models for both parsers have been optimised in the years that have passed since these shared tasks and thus the current results are slightly better for these parsers. During my work I have computed the attachment scores for all parsers mentioned in the Part I of the thesis. It is important to note that the results I have obtained might differ from what is reported in the literature, because I might have run the parsers not with the optimal parameters. Table 2 should give an impression how different parsers perform and how a CoNLL style evaluation looks like on the example of parsing the English test set.

According to this evaluation methodology the best parser is the one with the highest attachment scores.

5.2 Criticism

The traditional parser evaluation remains very popular up to the present day, because of the above mentioned advantages. However, it is not sufficient for

	UAS	LAS
Stanford Parser[6]	89.27	86.39
MST Parser	89.6	87.55
MaltParser(LibSVM)	92.1	90.4
MaltParser(LibLinear)	90.9	89.3
Ensemble	90.2	88.49
Mate-Tools	90.24	89.88
ClearParser	91.18	89.15
MDParser	89.7	87.7

Table 2: Attachment scores for dependency parsers

numerous reasons.

First of all, it covers only one aspect of parser quality, namely the accuracy with which it is able to replicate some gold standard. Some other important properties like efficiency, size of the data required to train the system and the usability of the system have not been a subject to evaluation. The efficiency evaluation would comprise the assessment of the training and parsing speed of a parser. The size of the required data is especially important for multilingual parsers working with resource-poor languages. Usability of a system involves the availability of preprocessing tools, different possibilities for the output formats, facility of feature model specifications, as well as the hardware requirements.

Additionally, as far as the accuracy evaluation is concerned, the tradition evaluation is not perfect neither. In fact there are numerous arguments why this kind of evaluation does not truly reflect the quality of parser's results:

- the parsers are tuned to a specific treebank, which might impair their performance for other domains, because they are overfit to some data and lose their ability to generalise.
- the scores do not differentiate between various dependency relations of different importance.
- the scores are high (~90%), suggesting that the results are very good, however, when used in applications, parser errors are omnipresent.

As far as another major advantage, namely the facility of parser comparison, is concerned the methodology also has some drawbacks:

- treebanks contain errors and a higher score when replicating the treebank might mean that there is not real improvement, but merely the errors are replicated.

- there are different annotation schemes, which vary in granularity. Some are easier to produce and some others are more difficult. E.g. the Stanford scheme is slightly easier than the CoNLL scheme.
- one parser might be better at more important and difficult relations, but they contribute equally towards the score as some other easier and less relevant dependencies.

Therefore a parser with a higher score not necessarily is better and a more detailed analysis is necessary.

5.3 Modified Evaluation

The disadvantages of the usual attachment scores practice lead to the necessity of a modified evaluation:

1. The problem of attachment scores being the average over many different relation types must be addressed.
2. The missing 10%, which can not be replicated, must be analysed and compared in order to determine whether they are common for different systems or whether the parsers have difficulties with various pieces of data.
3. The grade of tuning and overfitting of systems to a particular treebank must be assessed.

The first problem is well-known and there are numerous works dealing with it. The proposed solution is to define a set of relations relevant for one's needs and to compute attachment scores only over these dependencies. Examples of such work are the evaluation of non-local (long-distance dependencies, non-dependencies, phrasal modifiers and subtle arguments) [8] and unbounded dependencies (object extraction from a relative clause, object extraction from a reduced relative clause, subject extraction from a relative clause, free relatives, object question , right node raising and subject extraction from an embedded clause)[69]. Both papers provide a comprehensive motivation why exactly these types of dependencies are particularly important.

However, the two above-mentioned categories of dependencies are not relevant for one's needs in general. It is completely dependent on one's application what kind of relations are important and it can also be true that the simplest

ones are actually the relevant ones. Therefore to my mind no ultimate set of useful relations can be defined and that is why I was rather interested in developing a methodology of determining the set of relevant dependencies given a certain NLP task.

The notion of relevant or necessary dependencies has also already been discussed in several works. The basis for these works is the book by Anderson et al.: Entailment. The logic of relevance and necessity[6]. The general idea is that given some logic formulae and a task, relevant parts are those shared by the formulae, e.g. atomic units like variables and constants, whereas the necessary parts are those actually required to solve the task. E.g. for argument validation, the information contained in the premises should also occur in the conclusion in order to be relevant, but only some of such premises provide (or take away) support and are necessary in order to decide the validity. The same idea can be transferred to the information-based textual tasks: the relevant fragments are those which share the same information, whereas the necessary ones are those which are actually required for task purposes. E.g. in the shared task PETE (Parser Evaluation using Textual Entailments)[87] a necessary dependency is a dependency whose correct recognition is a requirement in order to correctly predict the entailment relation between two texts.

The exact distinction between relevant and necessary dependencies in a certain application is difficult and depends on how they are used in order to solve the actual task. However, it is obvious that at least as far as the evaluation of dependency parsing is concerned, it is very sensible to discard the irrelevant dependencies. Even though a further distinction between necessary dependencies and relevant ones would be a plus, the first step is already a big improvement.

The second point has also been addressed by several works. Especially the works on the integration of the transition-based and graph based approaches, as e.g. [66], discuss the differences between the results of various systems. These analyses have shown that different parsers perform differently for various parts of speech, dependency types and certain sentence constructions and that is why basically an integration of both makes sense. The Ensemble[77] system shows that even within one approach different parsing algorithms lead to different performance and that combining their strengths significantly improves the overall result. There is also an interesting article on Deniz Yuret's webblog[86][7] on the missing 10% problem. His belief is that a certain amount of the errors is due to

[7] http://denizyuret.blogspot.de/2010/11/next-generation-parser-evaluation.html, last retrieved May 10,2012.

inter-annotator disagreement, but does not perform an actual study.

The third point includes the efforts to study the problem of domain adaptivity for dependency parsing. The CoNLL 2007 shared task included two tracks: the multilingual track, which is the ordinary track where the training and test data for different languages were from the same resource, and the domain adaptation track, where the test data was different from what the parser was trained on. In terms of the domain adaptation track numerous works on the topic of domain adaptivity and overfitting were published. The finding was that the performance of the systems significantly dropped when they were applied to the out-of-domain data. The best approaches of that shared task, including those who achieved the top scores[32][75], used the knowledge about the new domain in order to improve the original model. However, it would be better to have a model which is as general as possible in the first place before adapting it to a different domain and this point has not been investigated. To my mind it is due to the fact that the community is not very aware of a considerable amount of erroneous annotations in the treebanks, which I have already broadly discussed in the Part I of the thesis. The overfitting is therefore dangerous, because a model tuned to such a treebank, might be simply replicating these errors.

This concludes the overview of the traditional evaluation methodology and the newer developments which are similar in nature. I will present my work in this field in section 7, but first I have to introduce the extrinsic evaluation methodology, which is on the other side of the spectrum.

6 Natural Evaluation

The traditional evaluation tries to evaluate the performance of a parser by assessing its ability to produce structures similar to what some gold standard suggests. However, for many people using parsers this is completely irrelevant. The parsers are usually used as a preprocessing step for some application and the only interesting question is therefore whether the parser results are beneficial for this task and to what extent. So instead of evaluating parser results directly the idea is to measure their contribution to solving some task for which parsing was applied.

There are numerous works which evaluate parsers in an embedding task, e.g. by incorporating dependency relations as statistical features in the task-specific system. Depending on the quality of the parsers the accuracy improvements, after including such features, will vary. Examples of this method include parser evaluation for information extraction[58][17][57] or textual entailment[87].

I am going to take textual entailment as example in order to discuss the advantages and disadvantages of the extrinsic evaluation.

6.1 Parser Evaluation with RTE

Textual entailment is a relation between text fragments, which states whether the meaning of one fragment is contained in the other one. The entailing text fragment is usually called text (T), the entailed fragment is usually called hypothesis (H), and both are usually referred to as T/H pair.

First, I will describe the idea of the already mentioned PETE shared task. The organisers of this task proposed a method for construction of T/H pairs for subsequent judgment whether T entails H or not. These pairs are constructed in such a way that the entailment relation can be predicted properly only in case when the necessary dependency relations were classified correctly. Here are examples for such dependency relations:

subject-verb dependency: "John kissed Mary." entails "John kissed somebody."

verb-object dependency: "John kissed Mary." entails "Mary was kissed."

noun-modifier dependency: "The big red boat sank." entails "The boat was big."

These examples show that entailment can be determined only if the parser correctly determined the subjects, the objects and the noun modifiers, respec-

tively. This is a very good additional possibility for evaluation of parsers, besides the usual evaluation metrics, since in most cases the main thing in real-word applications is to recognize the primary units, such as the subject, the predicate, the objects, as well as their modifiers, rather than the other subordinate relations. Overall this evaluation methodology offers the following advantages:

- Different dependency relations gain different importance: e.g. subjects and objects are more important than punctuation and determiners.
- The official results of the PETE challenge[8] show that parsers are far from being perfect and quite often produce errors which lead to wrong decisions.
- Different parsers and even different syntactic theories can be compared this way. E.g. it does not matter whether the subject is represented as SBJ, SUBJ, nsubj or any other specific label in order to match a certain gold standard, it is merely necessary that the entailment relation is determined correctly.

The T/H pairs used for evaluation were constructed manually and the organisers have invested a considerable effort in the creation of the data sets for the task, however, they have managed to provide the groups only with 32 T/H pairs for development and 301 T/H pairs for testing. This is definitely not enough in order to test parsers' performance for a large variety of dependencies. Due to the fact that the organisers tried to create only T/H pairs for which syntactic information alone is sufficient in order to determine the entailment relation, which is a very difficult task and which did not go smoothly in many cases(because beyond syntax logic, co-reference and semantics was required), it is very improbable that sufficiently large data sets will be created with this methodology.

Here are some of the pairs that require more than only syntax:

(4069 entailment="YES")

<t>Mr. Sherwood speculated that the leeway that Sea Containers has means that Temple would have to "substantially increase their bid if they're going to top us."</t>

<h>Someone would have to increase the bid.</h>

(7003 entailment="YES")

<t>After all, if you were going to set up a workshop you had to have the proper equipment and that was that.</t>

[8] http://spreadsheets.google.com/pub?key=tDUXEL8YSSL7S2vz_Jn_ckQ&single=true&gid=0&output=html, last retrieved May 18,2012

<h>Somebody had to have the equipment.</h>

(3132.N entailment="YES")

<t>The first was that America had become – or was in danger of becoming – a second-rate military power.</t>

<h>America was in danger.</h>

→ 4069, 7003 and 3132.N are examples for sentences where beyond syntactical information logic is required. Moreover we are surprised that sentences of the form "if A, then B" entail B and a sentence of the form "A or B" entails B, since "or" in this case means uncertainty.

(4071.N entailment="NO")

<t>Interpublic Group said its television programming operations – which it expanded earlier this year – agreed to supply more than 4,000 hours of original programming across Europe in 1990.</t> <h>Interpublic Group expanded.</h>

(6034 entailment="YES") <t>"Oh," said the woman, "I've seen that picture already."</t>

<h>The woman has seen something.</h>

→ In 4071.N one has to resolve "it" in "it expanded" to Interpublic Group. In 6034 one has to resolve "I" in "I've seen" to "the woman". Both cases are examples for the necessity of anaphora resolution, which goes beyond syntax as well.

(2055 entailment="YES")

<t>The Big Board also added computer capacity to handle huge surges in trading volume.</t>

<h>Surges were handled.</h>

→ If something is added in order to do something it does not entail that this something is thus automatically done. Anyways pure syntax is not sufficient, since the entailment depends on the verb used in such a construction.

(3151.N) <t>Most of them are Democrats and nearly all consider themselves, and are viewed as, liberals.</t>

<h>Some consider themselves liberal.</h>

→ One has to know that the semantics of "consider themselves as liberals" and "consider themselves liberal" is the same.

6.2 Criticism

Apart from the difficulties of creating appropriate data sets in order to test applications on them, natural evaluation has some additional drawbacks. Since it is not the parser accuracy being evaluated, but the NLP system. Therefore it is important to have a good module for making use of parser results in order to predict the entailment relation correctly, and this is not a trivial task. For RTE systems, if T entails H, some dependency relations of H have to match those occurring in T. However, sometimes the matching is not straightforward like subject(T) = “John” and subject(H) = “John”, but can be trickier like object(T) = “Mary” and object(H) = “somebody” or even subject(T)=”I” and subject(H)=”woman” (in T = "Oh," said the woman, "I've seen that picture already."; H = The woman has seen something.)[9] Additionally, when applying to real-world data in most cases synonymy and/or semantic relatedness becomes important, such as in the following T/H pair:

T = Pet owners were forced to abandon their animals in the midst of evacuation.

H = People were forced to leave their pets behind when they evacuated New Orleans.

Here the following dependencies have to match: subject(T) = “pet owners” and subject(H) = ”people”, as well as vc(T) = “abandon” and vc(H) = “leave” and object(T) = “animals” and object(H) = “pets”.

Furthermore, the problem with this method is that different T/H pairs can have completely different levels of difficulty, e.g. as far as the number of correctly recognised relevant dependencies is concerned, but they contribute equally to the score. Finally, sometimes the correct decision is made simply by chance (only two possible classes YES and NO, with good chances of guessing) despite the wrongly predicted dependency structure.

My participation in RTE-6[80] shows that for real-world textual entailment data of the RTE-6 challenge it is very difficult to achieve a high f-score using syntactic dependencies, not only because they might be incorrectly predicted by the parsers, but because the module for matching becomes too complex, since all sorts of knowledge, including, but not limited to lexical semantics, coreference resolution, logic and inference, world and domain knowledge become necessary.

Thus, the natural evaluation is not flawless neither. Even though different parsers can be easily interchanged in the same system and their impact on the

[9] All examples are real examples from the PETE shared task.

final result can be compared, this strategy has the following drawbacks:

- It is unclear whether the impact is due to parser quality or due to the quality of the embedding, and what is the relation between both.
- Depending on the task the embedding might be very difficult, both time-wise and regarding the required level of expertise.

6.3 Modified Evaluation

The disadvantages of the extrinsic evaluation lead to the following requirements of a modified evaluation:

- The embedding should be simple and it should not favour any parser, so that a change in performance could be attributed only to the quality of the parser results.
- It should be clear whether a low result of the system is due to the difficulty of the task or simply because the system performs poorly. Similarly it should be obvious whether a small difference in the system's result with parser compared to the result without parser is due to the fact that parsing does not help solving the task or because the parsing is too inaccurate in order to contribute to the solution.

On the contrary to the traditional evaluation which has a lot of proponents and a lot of literature concerning its problems, extrinsic evaluation is much less studied. The reason for that is the already mentioned disadvantage that such embeddings are difficult and time consuming, because an extrinsic evaluation requires several steps to be done.

First, it must be investigated whether one's task can profit from parsing at all. Only then, in case the the latter is true, different parsers can be applied. Both steps are not trivial, and many studies already end after the first phase.

The investigation whether parsing is beneficial for one's application is usually done via adding some features based on the parser's output to the original training data and measuring the difference between the new result and the previous performance. However, such features are sometimes very parser-specific and do not work well if a different parser is used, even if the parser uses the same representation. The latter is actually quite rare and since most parsers use different representations and either an intermediate common representation or a series of parser-specific solutions are necessary if one wants to compare them.

Attempts to investigate whether dependency parsing is beneficial for a certain NLP application has been undertaken in almost every area: e.g. machine translation[73][83], POS tagging[47], RTE[70], question answering[24] or relation extraction[16]. However, all these works are limited to the first step, i.e. only one parser is tested. The literature for extrinsic parser evaluation is much more sparse. E.g. the already mentioned work by Miyao et al.[58][57] for information extraction, the comparison of two parsers for question answering[59] or investigation of different parser appropriateness in the context of human robot interaction[45].

However, all these works show that both ways of comparing parsers are problematic. If some common representation is chosen to which the outputs of different systems are transformed in order to guarantee their comparability, then the extrinsic evaluation becomes a three-step evaluation: a) the parser has to be evaluated b) the embedding of the parser results has to evaluated c) the transformation of the parser results to the common representation has to be evaluated. Otherwise, if parser-specific solutions are done the evaluation also gets an additional uncertainty in it, namely whether one parser-specific solution is not better than a solution for another parser, which might distort the results.

This concludes the overview on the extrinsic evaluation. I will now present my work in the field of parser evaluation, which attempts to cope with the disadvantages of both evaluation methodologies.

7 MDParser Evaluation

In this part of the thesis I have so far introduced two possible approaches for dependency parsing, as well as discussed their advantages and disadvantages. I have also given an overview of the most prominent works in this field. In this section I am going to present my work in the field of dependency parsing evaluation, which mainly had the following objectives:

- I wanted to experiment with both evaluation methodologies myself, in order to better understand the difficulties of both from my own experience.
- I wanted to evaluate MDParser and compare it with other state of the art parsers
- I wanted to develop a methodology which avoids the disadvantages of the intrinsic and extrinsic evaluation approaches.

In this subsection I am going to evaluate MDParser, both with the intrinsic and extrinsic evaluation methodologies.

7.1 Intrinsic Evaluation

I have already presented an overview of attachment scores for all main state-of-the art parsers in Section 5.1 (Table 2). We see that MDParser is in the lower range segment as far as the accuracy is concerned, however, it is not the worst performing system. The gap between the top performing parser MaltParser(LibSVM) and MDParser is 2.4%.

Even though accuracy is not the only important property of a parser and I will come to this point later, in this subsection I am going to focus on accuracy and its importance. MDParser has always been inferior in terms of accuracy to other parsers in the field. Therefore I tried to analyse how significant this gap of few percents is. Therefore I will compare MDParser with the best performing parser MaltParser.

The main difference between MaltParser(LibSVM) and MDParser is the machine learning strategy used for training their models and predicting the transitions. As the name suggests MaltParser(LibSVM) uses LibSVM and MDParser uses LibLinear machine learning libraries. However, because there are also other differences between MDParser and MaltParser, as I have already broadly discussed in Section 3 of the thesis, I thought that in order to compare LibSVM and

LibLinear it is better to use MaltParser with both libraries and study the differences. Thus the first step in my investigation of the gap between MDParser and MaltParser (LibSVM) is to study the differences between both machine learning strategies, which are mainly responsible for the different accuracies achieved by the systems.

As already mentioned, MaltParser(LibSVM) achieves a LAS of 90.4 and MaltParser (LibLinear) achieves a LAS of 89.3. A detailed analysis shows that 86.9% (29015 tokens) of all dependencies are equally recognised by both models and are correct. 2.9% (968 tokens) are correctly recognised only by LibSVM model and 1.7% (577 tokens) – only by LibLinear, the rest is not recognised by neither of both. I have performed a detailed analysis of the 2.9% of dependencies which are only recognised when using LibSVM. The analysis consisted out of the following steps:

1. Investigation whether these 2.9% are just an ordinary extract of the data, which contains approximately the same dependencies as any other sample section of the data.

2. Investigation whether LibSVM performs better simply due to overfitting the data.

3. Investigation whether quadratic kernel, being a more complex approach, helps to recognise more difficult dependencies better than the simpler linear SVMs.

Therefore I have done the following three experiments:

1. I have computed the distribution for different dependency types for the whole data and for the 968 tokens, which are only correctly processed by LibSVM, and compared them. Most dependency types indeed have approximately the same distribution as the entire data. These dependency types account for 61.8% of the 968 tokens. However, the rest is quite different and occurs either much more or much less often, i.e. at least by a factor of 2, than according to the distribution of the whole data. E.g. the rather unimportant dependency type P (punctuation) accounts for 11.02% of all dependencies in the English data. In these 968 tokens punctuation occurs 253 times, i.e. 26.13% of all dependencies. Among other dependencies which occur with an above-average frequency were tokens of type COORD (coordination), which occurred 65 times (6.71%),

whereas the average is 2.65%, and APPO (apposition) - 32 times (3.30%), average 1.41%. The above-average of correctly recognised coordinations, punctuations and appositions gives rise to the suspicion that LibSVM is particularly suitable for longer sentences, with relative clauses and appositions, as well as punctuation associated with these phenomena. Indeed, our experiments showed that whereas the average sentence length of the sentences in the data is 25.01 words, the average length of the sentences, which are correctly parsed by LibSVM but not LibLinear is 30.67.

2. Any treebank contains a certain amount of inconsistency due to the nature of its creation. The already presented work on treebank errors shows that around 1% of all annotations in the English treebank is erroneous. The idea was to test whether the number of inconsistencies is higher for the 2.9% of data we are investigating. After applying their approach to this portion of the corpus we were able to identify 86 errors (8.3%), which exceeds the expected average by several times. That means that actually a big portion of what is recognised as correct by LibSVM and not LibLinear is actually due to erroneous gold standard. A similar experiment for LibLinear revealed that it only classified 41 tokens with erroneous gold standard. Thus LibLinear seems to be able to generalise better, whereas LibSVM merely replicates the gold standard, even if it is inconsistent.

3. Whereas many parsers achieve results around 90%, the score is merely an average over all dependencies. Among them are those which are rather easy and unimportant for applications, and those which are difficult and highly relevant for certain tasks. The performance of parsers for the latter dependencies is usually much lower. Examples for such dependencies are non-local dependencies and unbounded dependencies. I have taken the data with non-local dependencies and evaluated MaltParser(LibSVM) and MaltParser(LibLinear) on it in order to investigate whether there is some difference between the models for more difficult phenomena. The result has shown that there is absolutely no difference between both parsers and they were able to recognise exactly the same number of demanded annotations.

This explains the main difference between MDParser's and MaltParser's accuracies. However, there is still a difference of around 1% between MaltParser (LibLinear) and MDParser, even though both parsers use the same machine

learning strategy. I have experimented with all possible parser parameters in order to find the reason for this difference and was able to determine that my feature model is responsible for that. The dependency label features 14-17 (see Section 3.2) in MaltParser are very important and contribute to a big portion ($>$ 1%) of its accuracy, whereas in MDParser they are merely responsible for a very slight improvement (~0.2%). When MaltParser and MDParser are trained without these features then the accuracies of both systems are virtually the same. Unfortunately, I was not able to find out the difference in the usage of these features and why MDParser can not profit from them as much as MaltParser does.

Overall, this intrinsic evaluation has shown that a higher score of the parsers using more sophisticated machine learning techniques is due to the better ability to replicate the gold standard. However, this does not really mean a better quality of the results, since they have a more tuned model and thus also blindly replicate inconsistencies. The results also exactly the same for ten non-local dependency types. Only a small difference in accuracy is due to some technical imperfectness of MDParser.

7.2 Extrinsic Evaluation

I have undertaken several attempts to extrinsically evaluate MDParser. First, I have participated in the PETE shared task, which was designed particularly for the purpose of parser evaluation. I have already introduced the PETE task in Section 6.1.

For the PETE challenge I have built a system which compares the output of a dependency parser for T and the output for H in order to predict the entailment relation for a T/H pair. The comparison consists of a set of rules which are used to judge the similarity of T and H (cf. [81] for more details). I have used four parsers in order to construct these results: MDParser, MaltParser(LibSVM), MiniPar and StanfordParser. Even though I have used a common representation, to which I was able to transform the results of all parsers I have tried out, it turned out that the parsers produce different structures and that my rules favoured the structures of MDParser and MaltParser, which are the same and which were initially used when the PETE system was developed. StanfordParser does not only use a different tagset, which in this case was not very problematic, because of a very limited number of dependency types which were important for the entailment relation, but it also assumes a different tree structure[28],

	Accuracy
MiniPar	45/66
Stanford Parser	50/66
MaltParser	51/66
MDParser	50/66

Table 3: Parser Comparison for the PETE Development Data

	Accuracy
MaltParser	196/301
MDParser	197/301

Table 4: Parser Comparison for the PETE Test Data

which would require a completely different set of rules. MiniPar is a rule-based system, also with a different set of dependency relations, which additionally sometimes constructed either partial or no structures at all, when the input was not covered by its grammar, e.g. because it was ungrammatical in some cases. Therefore an automatised evaluation of all four parsers was impossible within the scope of the challenge.

I have performed a manual evaluation of these four parsers for the PETE development data (66 T/H pairs):

For the test data such manual evaluation was impossible, because there were too many T/H pairs (301 pairs). Therefore I have restricted my evaluation to MDParser and MaltParser, for which the entailment prediction was automatised.

The PETE challenge shows that despite the fact that MDParser achieved an inferior attachment score it is absolutely suitable for the PETE data and achieves the same results as some other more sophisticated systems.

I have already mentioned in Section 6.1 that despite the effort neither the quantity nor the quality of the PETE data was satisfactory and that it was unprobable that bigger and better data sets will be created with this methodology. In order to avoid the data sparseness problem I have therefore decided to try out a parser comparison with textual entailment on real-world data. For that purpose I have taken part in RTE-6 challenge[9], which offered a lot of real-world data with textual entailment annotation, and have applied the approach from the PETE task in order to compare MaltParser and MDParser. The resulting ranking of the parsers, as well as the difference between the individual results, corresponds perfectly both to the LAS/UAS and PETE comparison be-

	F-Score
MaltParser	39.81
MDParser	38.26

Table 5: Parser Comparison for the RTE-6 Data

tween these parsers, which makes the idea of evaluation parsers with textual entailment more feasible since the experiment shows that no artificial syntax-restricted data is necessary.

RTE-6 task offered large sets of data both for development (around 15,000 T/H pairs) and testing (around 20,000 T/H pairs). Of course, the real-world data of RTE-6 requires more than syntax in most cases. However, I have performed an analysis of a big portion of positive T/H pairs in order to determine what type of knowledge is required in order to correctly classify them. I have tried to group all instances into three classes: **A** - syntax, **B** - lexical semantics and **C** - inference.

As an example let us take one hypothesis H_1 and several different T_i that entail H_1. Depending on the type of knowledge required to infer the entailment relation the Ts can be split into different classes:

H_1: People were forced to leave their pets behind when they evacuated New Orleans.

A: T_1: Thousands of people were forced to leave their pets behind when they evacuated New Orleans.

B: T_2: Animal rescue officials have been collecting scores of pets and other animals from the shattered city, while many survivors have told of their distress at having to leave beloved cats and dogs behind in the watery city when they fled.

T_3: Such emotional scenes were repeated perhaps thousands of times along the Gulf Coast last week as pet owners were forced to abandon their animals in the midst of evacuation.

C: T_4: For Elizabeth Finch, the owner of two dogs named Zorra and Hans Blix, the sight of citizens forced to choose between their pets and their safety was, like the disaster itself, indicative of broader social rifts.

T_5: The animals are being cared for at a farm north of Louisiana until they can be reunited with their families, many of whom were told they would not be able to bring their pets on evacuation buses and helicopters.

The class **A** is the easiest one - the relevant information is expressed with

the same words in both T and H. The maximum that should be done in this case is the analysis of the syntactic structure in order to determine that the structure of T contains the structure of H and thus T entails H, cf. T_1/H_1 pair. The class **B** is a little bit trickier, e.g. the words used in T_2 and T_3 differ from those used in H1. Thus in order to correctly recognise the entailment relation one has to know about the synonymy of the words animals and pets or leave behind and abandon in addition to the syntactic structure. The class **C** is the most difficult one. For that class of T/H pairs one has to use logic inference and/or world knowledge. For example in order to imply H_1 from T_5 one has to know that New Orleans is in Louisiana. For the pair H_1/T_4 some deeper logic inference is required in contrast to the rather simple predicate matching of the **A** or **B** classes.

According to my findings around 30% of all positive T/H pairs belonged to the class **A**. Even though this is only a portion of the whole data it is still considerably more than the PETE task could offer. With this experiment I could show that picking syntax-based T/H pairs from real-world data is a better option than trying to artificially construct such pairs.

7.3 Combined Evaluation

Having analysed the strengths and weaknesses of the existing evaluation methods I have decided to develop an alternative, combining their positive aspects and avoiding their disadvantages. The following method has been published and is to appear at an IEEE 2012 workshop EMRITE.

On the one hand I thought that it is essential for the evaluation to be task-specific, since it is a perfect possibility to find out whether a parser is suitable for the given task or not. At the same time, I believed that it is a great idea to restrict the evaluation set of dependencies only to the important ones. However, to my mind there is no universal set of important relations, because for one task one set of relations might be relevant and for another task it could be a completely different one. Additionally, I wanted to avoid the embedding into a broader NLP application context, typically done in extrinsic evaluation, since then it becomes difficult to differentiate between the quality of the dependencies and the quality of the embedding.

The resulting methodology looks like this:

1. Identify the relevant tokens (words) for the given task (cf. Yuret et al.[87] with the necessary dependency relations for recognising textual entail-

ment).

2. Annotate these tokens with the desired dependency relations.

3. Parse the data.

4. Compare the output of the parsers with the manual annotation.

The proposed methodology is thus a combination of intrinsic and extrinsic methods. On the one hand it is a task-specific evaluation, however, instead of embedding the parsers into an application the evaluation takes place on the level of grammatical relations. Additionally, only the important tokens are evaluated and therefore the overall score is not distorted by the average for all tokens. The obvious disadvantage is that the annotation has to be done manually and in theory it requires the knowledge of the dependency grammar representation. However, in practice the overwhelming majority of dependency relations between the relevant tokens is of simple nature, since they belong to such easy-to-annotate types as subjects, objects or modifiers. In any case, from our experience, since we have done both in our former work, the task of annotation requires much less expertise than the task of embedding of dependency relations into an NLP system. Furthermore, the annotation process can be semi-automated, e.g. by initialising the annotation of all identified tokens with the one proposed by some parser. The actual annotation process is then reduced to the manual correction of the latter, which is usually much less work than providing the annotation from scratch.

I have applied this methodology for a small part of RTE-7[10] development data. I have processed 100 positive T/H pairs (from 1136 total). For these 100 pairs I have taken the corresponding 100 hypotheses and applied the algorithm. I took positive pairs, because they always overlap in meaning, on the contrary to the negative pairs which sometimes were completely unrelated to each other. Because the negative pairs account for the overwhelming majority (>95%) it would have unnecessarily complicated the annotation process, especially because I could not even annotate all the available positive ones. I did not apply the strategy to both texts (Ts) and hypotheses (Hs), but rather only hypotheses, since both T and H of the same T/H pair usually contain very similar dependencies and it would require the double effort in order to obtain the double amount of approximately identical material.

Furthermore, it is important to note that Hs could not be taken independently of the T/H pair they occur in, since the set of relevant tokens in H

depends on the particular T. E.g. consider the following H = "Christine O. Gregoire has been elected Governor." This H is entailed by the following Ts: T_1 = "Christine O. Gregoire, the Democratic attorney general, last week was declared the winner."; T_2 = "But for now, Gregoire remains scheduled to take the oath of office, and she insists she will do so."; T_3 = "Fifty-eight days after the election, Christine O. Gregoire was declared the governor-elect of Washington on Thursday".

When identifying the relevant tokens in H in combination with T_1 one does need the information that it was a gubernatorial election. For T_2, in addition to that, the first name becomes irrelevant. In contrast to that, for T_3 all tokens are relevant.

Overall, the analysed 100 hypotheses consisted out of 1058 tokens. 664 (62.8%) of them were marked as relevant. Eventually, the 664 tokens were annotated with a manually created gold standard and then compared with the results produced by MST Parser, MaltParser and MDParser:

	LAS
MST Parser	88.4
MaltParser	85.6
MDParser	85.6

Table 6: Parser Comparison for Relevant Dependencies

The relevant dependencies were of the following types:

Dependency Type	Count
OPRD	10
NAME	48
LGS	7
IM	7
TMP	9
AMOD	3
OBJ	34
DIR	2
SBJ	99
ADV	25
DEP	2
LOC	25
PMOD	88
VC	36
CONJ	2
SUB	2
PRD	42
COORD	3
MNR	1
ROOT	87
APPO	9
NMOD	123

Table 7: Relevant Dependency Types

The evaluation shows some interesting facts. First, the MST Parser and MaltParser, which achieve almost identical results for the standard CoNLL test data perform differently for the RTE-7 data or at least for the fraction that I have selected. Additionally, the evaluation shows that MDParser is no longer inferior to other parsers for this task. Thus the method helped to find a more suitable parser for the task, where the traditional evaluation would not suffice. Second, the analysis demonstrates that only half of the data is relevant and requires a correct dependency analysis. It does not matter how the parser performs for the rest of the data. As shown in Table 7, the relevant relations belong to a very small subset of all relations present in the data (overall there are more than 40 different types), which once again emphasises that averaging parser performances over all tokens is not appropriate. The most important ones are the main predicates of the sentence (ROOT, VC or PRD), as well as subjects, objects, locations and modifiers. These relations should thus be prioritised.

I have also performed several experiments on the relevant dependencies in

the CoNLL test data:

First, I have examined how many dependencies belong to the set of relevant ones. Only ~73% of all dependencies are of one of the relevant types, which again confirmed our argument that an average over all tokens is not appropriate.

Second, I have evaluated the performance of MaltParser and MSTParser only for the relevant types and found out that a) MaltParser performed better (92.5% LAS MaltParser, 91.7% LAS MST Parser, 91.9% LAS MDParser) for the CoNLL data and b) the performance is higher than the average over all dependencies (it is around 90% for both parsers). Both points also support our thesis that an evaluation on a different domain is not transferable to the desired application domain: a) demonstrates that the traditional CoNLL evaluation on the standard test data would not help selecting the most appropriate parser for the task. The point b) additionally demonstrates how the performance of a parser drops as soon as the domain of the application is not the same as the one the parser was trained on and that despite the fact that the relevant dependencies even seem to be easier than average, because of the higher scores for them compared to the overall score over all dependency types.

The most problematic part about this approach is the determination of what is a relevant or a necessary token. It is quite easy in case of the PETE shared task data, where each T/H pair aims at evaluation of only one necessary dependency relation per T/H pair and the same words in both T and H are used. However, for real-world Ts and Hs, selected out of newspaper texts, it is much more difficult, because in most cases numerous dependencies expressed with different words are necessary. However, as I have already introduced the notions of necessity and relevance in Section 5.3, while it is difficult to differentiate between necessary and relevant tokens, because what is necessary often depends on the approach for solving a certain task, it is already a big improvement to discard the irrelevant dependencies from the evaluation, which is usually not so difficult.

7.4 Additional Evaluation

So far, both intrinsic and extrinsic evaluation always consisted of assessing the quality of parser results. Whereas it is definitely an important, probably even the most important, property of a parser, a decent evaluation involves much more than this. Some other properties like efficiency, size of the data required to train the system and the usability of the system are also very significant, especially for application-oriented parsing.

MDParser	0.0008 seconds / sentence
MSTParser	0.268 seconds / sentence
MaltParser (LibSVM)	0.265 seconds / sentence
MaltParser (LibLinear)	0.0025 seconds / sentence
Ensemble	0.01 seconds / sentence
Mate Tools	0.077 seconds / sentence
Stanford Parser	0.37 seconds / sentence
ClearParser	0.0029 seconds / sentence

Table 8: Efficiency Evaluation

7.4.1 Efficiency

The quality of the result is not the only requirement parsers should meet. Many other applications, especially those which work with huge amounts of data or applications where processing has to be done online within milliseconds, require parsing to be particularly fast in order to be eligible for use. This has often been neglected, because parsing is usually done on rather small test sets, where the parsing speed does not play any role. For MDParser it has always been the top priority to be fast, so that it can also be used in applications which work with the web. Web applications, which usually demand processing of thousands of sentences require fast parsing and the quality of results become secondary.

Here is the overview of the results of different state-of-the art parsers for the English test data:

The evaluation shows that MDParser is 4 times faster than the second best parser MaltParser (LibLinear) and is able to parse a sentence in less than a millisecond. It is hundredfold faster than most other systems.

For StanfordParser, MSTParser, MaltParser and MDParser I have used a machine with 2,4 GHz processor with only one core used. The information about the efficiency of other parsers is taken from the respective literature: Ensemble[77], Mate Tools[12] and ClearParser[21].

7.4.2 Size of the Training Data

Annotated resources are expensive and difficult to create. Therefore, for many languages with smaller NLP communities resources like treebanks are rather small. For that reason it is important to know whether a certain system, which performs well for a language like English would also perform well for one's own language, which might not have such a big amount of annotated data.

In Section 4.2 I have already described the experiment I have performed in order to investigate whether LibSVM and LibLinear versions of MaltParser (and thus also MaltParser (LibSVM) and MDParser) require different size of training data in order to achieve high accuracy and thus would influence the choice of the machine learning library for languages with small dependency treebanks. Figure 5 shows that there are no difference between the different versions of SVMs.

For this subsection I have also performed such an analysis for MST Parser, since besides transition-based parsers, which are based on MaltParser, many other graph-based parsers are based on MST Parser. I have performed the same evaluation for it, i.e. I have trained 20 models for different sizes of the treebank. Whereas MaltParser (LibSVM) or MaltParser (LibLinear) / MDParser are able to achieve the accuracy merely 1% below their top one already with 40% of the data, MSTParser requires 55% of the data for that. Thus the SVM-based transition-based dependency parsers are better suited for processing resource-poor languages than the graph-based parsers.

In the past, less sophisticated machine learning methods with faster training usually required much more data than more sophisticated support vector machines and their usability was thus limited to resource-rich languages. This evaluation, however, shows that MDParser does not require more data than other state-of-the-art parsers and is thus suitable for multilingual dependency parsing.

7.4.3 Usability

The performance of a system is irrelevant if a parser is difficult to use and/or is not adoptable for one's own application. The usability of a system involves the availability of preprocessing tools, different possibilities for the output formats, facility of feature model specifications, as well as the hardware requirements.

First, let us address the importance of the preprocessing tools. Since parsing is often treated as an independent task, it is often neglected that it requires a lot of preprocessing in order to work properly. The necessary preprocessing steps include: sentence splitting, tokenisation and POS tagging. The only parser which comes with all these preprocessing tools is Stanford Parser and thus it is the only parser which can be used for processing plain text out of the box. All the other parsers would require additional tools to bring the text into the desired format first. Stanford Parser is a very popular parser in the community,

despite the fact that it is neither accurate nor fast, compared to the more recent developments. Thus the fact that it is applicable to plain text in such an easy fashion shows the importance of this property to the people who use parsers.

Therefore, MDParser also supports all preprocessing steps. For preprocessing I have chosen components which are multilingual and fast. E.g. for sentence splitting and tokenisation MorphAdorner[1] package is used. It is able to automatically recognise the language used and process the input accordingly. Since it based on ICU4J[2], a set of libraries designed to work with unicode, it is able to work with an arbitrary language, as long as the input is in unicode. As far as POS tagging is concerned, MorphAdorner unfortunately uses a different set of POS tags from what is typically used in dependency treebanks. Therefore I have used a different POS tagger, which has been developed by my colleague at DFKI Sven Schmeier. Even though it is not as flexible as MorphAdorner, its models are learned from dependency treebanks and thus it is guaranteed that for any language for which MDParser is to be applied it is possible to perform the POS tagging. As far as their performance is concerned the tools are very accurate and fast. Whereas sentence splitting and tokenisation are in general relatively inexpensive tasks, POS tagging might be a bottleneck in a complex system. However, the POS Tagger used in MDParser is particularly fast because it is based on fast SVMs[35].

Plain text is often not in the same format as the system expects it to be, especially as far as special characters are concerned, which might deteriorate the overall performance. A typical user, however, does not know the requirements of the parser and if he or she should have to select the preprocessing components, they might not provide the desired results. A great advantage of integrating all components in one system, besides the fact that it is simply more convenient for the user, is thus the guarantee that they produce exactly the output expected by the parser.

The second important issue are the output formats. Before one can use the result of a parser, the result itself usually has to be read in and analysed. E.g. for phrase structure parsers the difficulty lies in the rich bracketing and its interpretation. For dependency parsing the problem is usually that dependency relations are binary and sometimes a relation between some important words in a sentence is not direct, but involves some intermediate tokens. E.g. the relations between two content words sometimes involve various prepositions, which makes the usage of the results in applications more difficult, because patterns become more diverse. Another problem is that in order to encode the

dependency relations only the indexes of the corresponding words are used. In case one often needs word forms and POS tags of the corresponding words a lot of lookups might be necessary with this type of representation. For these reasons many parsers provide their own somehow special output formats, which sometimes makes it difficult to replace a parser in one's application because the output of some new parser would not fit in the system.

MDParser supports many different output formats which are useful for various reasons. One of its most useful properties is that both Stanford and CoNLL output formats are supported and thus any system which had been using some other parser before can be easily tested with MDParser and the difference in the performance can be measured.

The third point concerns feature model specification. In most systems feature models can not be changed easily. This has the drawback that when switching to a new language usually some default model is used. The only notable exception is MaltParser, which provides a special feature specification language which allows to define feature models in an XML file, without having to modify the code.

MDParser does not allow easy feature model specification, i.e. feature models have to be specified in the code. The reason for that is that features are often quite arbitrary and a special language can not foresee what kinds of features a user wants to try out. The feature specification language of MaltParser is not an exception and has rather limited expressive power, even though a lot of effort has been invested into its creation. To my mind, most users use default models anyways and those who wish to experiment will be very quickly limited by the expressive power of the specification language. However, a truly user-friendly system definitely should provide some sort of solution to this problem, since altering of the feature model without recompilation is doubtlessly a useful feature of a parser.

Finally, the hardware requirements also differ greatly among different parsers. MST Parser and MaltParser (LibSVM) require 64 Bit machines with a lot of memory (usually >4 GB, up to 20-30 GB for large treebanks) in order to train and use their models.This might be difficult for people who want to run the parser on their private computers, which are usually not so powerful. Especially, if one considers that the parser has to be additionally embedded into some application which has its own requirements, then this property becomes even more important. On the contrary to that LibLinear not only allows fast training and application, but it also has much more modest requirements so

that much less memory is necessary and it can usually be run on ordinary computers. E.g. for the very big English training data MDParser only needs 1 GB of memory.

8 Conclusion

In this thesis I have broadly described the state of the art dependency parsing. I have discussed the major problems with the current situation, very most that parsing is seen as an independent task, without considering the preprocessing steps before and usage in applications after it. I have analysed dependency parsing systems and their properties, presented the most prominent state-of-the-art parsers and my own development - MDParser. I have motivated the creation of yet another dependency parser and described the work I have undertaken in the years I have been working on the thesis. A lot of this work was unsuccessful or became obsolete in the course of the years, however, there are also a considerable amount of successful experiments with interesting results, which I have presented in this thesis.

Additionally, I have addressed the topic of dependency parsing evaluation. Here, I have also described the state of the art and pointed the one-sidedness of the current evaluation which focuses on the accuracy too much and neglects other system properties, which are especially important for parsers used in applications. I have presented both evaluation methodologies used in the field: the traditional evaluation, which is used in most cases and reflects the percentage of structure successfully constructed by a parser, as well the natural evaluation, where the contribution of the parser for solving some task is measured. For this topic I also present some successful experiments on avoiding the weaknesses and combining the strengths of both methodologies. I perform a thorough comparison of my parser with other state-of-the-art systems.

The goal of the thesis, was to study how to select the most appropriate parser for one's application. I have done an elaborate analysis of the current situation in the field of dependency parsing, implemented my own parser, participated in shared tasks and helped people to use dependency parser results in applications. This experience has clearly shown to me that the best parser is definitely not the one with the highest accuracy, but there is more to it, e.g. the ability to process the plain text, efficiency and usability. The task-oriented evaluation strategy I have proposed in this work can also help a lot in achieving this goal.

In the course of the work MDParser developed into a very mature and usable system. To my best knowledge it is currently the fastest dependency parser available and by the end of the development it was almost able to reach the top accuracies of the other systems. It is the first data-diriven parser since Stanford Parser which is able to process plain text and does not require any additional

preprocessing before it can be applied. Since Stanford Parser is already quite outdated as far as both accuracy and efficiency are concerned it is also a great achievement of this work. Moreover, MDParser does not require as much data as some other state-of-the-art parsers, its models are compact and do not require much space and it does not need as much resources to be run as it was usual for many other systems. When I was able to achieve this major milestone and create such a fast, but still accurate parser, I have decided to finish my "PhD project". However, for time reasons I still have not explored some important aspects of dependency parsing.

First, MDParser heavily relies on machine learning and, as I have already mentioned, I have had to replace many machine learning packages by newer and better ones in the course of the development. For every learning strategy I have intensively experimented with many languages, usually including English, German and some other resource-poorer languages, which always had been a time-consuming work. However, for the final machine learning strategy I have not yet experimented a lot with other languages, because otherwise I would risk that my results become outdated again. That is why I have focused only on English in this thesis. Of course, multilinguality is of an extreme importance for application-oriented parsing and therefore it is certainly a shortcoming of my work, however, on the other hand I do not expect that the results of MDParser would be different for other languages and the general picture would change somehow, as it has not been the case neither with all the other machine learning packages I have tried out in the past.

Second, I have neglected projectivity in my work. The reason is that I was always doing application-oriented parsing, i.e. the primary objective was the correct recognition of dependencies, which are relevant for some application. However, non-projective dependencies are rare (even for the languages which have a lot of non-projective the total amount is around 2% and it is much lower for most other languages) and they are even rarer among the relevant ones. The extrinsic studies which I have performed confirmed this assumption, since no relevant dependencies were non-projective in the tasks I have investigated. Therefore, especially for a language as English, it is usually much more sensible to risk losing some non-projective dependencies instead of increasing complexity of the whole system, when one does not only have the highest possible accuracy in mind. However, a good parser, especially if it is to be used for many languages, among which certainly also those with a lot of non-projectivity might occur, should have a decent solution to this problem.

Finally, the recent developments show that parallel computing is growing in importance, because any machine nowadays has several processor cores. Parsing is extremely suitable for parallelisation, because many tasks are independent from each other and can be done simultaneously. Experiments show that a considerable improvement in speed can be achieved by doing parallel feature extraction or by dividing the input into pieces, which are then delegated to different CPUs for processing. Accuracy can also be improved, e.g. by running the system with different settings (e.g. models, algorithms) in parallel and then combining their results. Thus, especially since I was interested in fast parsing, the most obvious thing would be to use parallel computing. However, from the scientific point of view the acceleration of the processing by the means of brute computing power is not very interesting. Technically, it is also a very easy task, since something like splitting the data into pieces and calling an instance of the parser for each piece is not very challenging neither. Therefore I have decided not to focus on parallel computing and/or improving the speed by pure technical means. However, in the end, application-oriented parsing requires very high efficiency and it is then does not matter how it is achieved. Thus it is important to make use of several CPUs if available, which MDParser unfortunately does not do yet.

These are definitely some of the directions I have to pursue in the future in order to make MDParser even better.

References

[1] http://morphadorner.northwestern.edu.

[2] http://site.icu-project.org/.

[3] S. Abney, S. Flickenger, C. Gdaniec, C. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski. Procedure for quantitatively comparing the syntactic coverage of english grammars. In E. Black, editor, *Proceedings of the workshop on Speech and Natural Language*, HLT '91, pages 306–311, Stroudsburg, PA, USA, 1991. Association for Computational Linguistics.

[4] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic parsing of ambiguous grammars. *Commun. ACM*, 18(8):441–452, August 1975.

[5] A. Aizerman, E. M. Braverman, and L. I. Rozoner. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[6] Alan Ross Anderson, Nuel D. Belnap, and J. Michael Dunn. *Entailment. The logic of relevance and necessity. Vol. II.* Princeton University Press, Princeton, NJ, 1992. With contributions by Kit Fine, Alasdair Urquhart et al, Includes a bibliography of entailment by Robert G. Wolf.

[7] Collin F. Baker, Charles J. Fillmore, and John B. Lowe. The Berkeley FrameNet project. In *COLING-ACL '98: Proceedings of the Conference*, pages 86–90, Montreal, Canada, 1998.

[8] Emily M. Bender, Dan Flickinger, Stephan Oepen, and Yi Zhang. Parser evaluation over local and non-local deep dependencies in a large corpus. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 397–408, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[9] Luisa Bentivogli, Peter Clark, Ido Dagan, Hoa T. Dang, and Danilo Giampiccolo. The sixth PASCAL recognizing textual entailment challenge. In *The Text Analysis Conference (TAC 2010)*, 2010.

[10] Luisa Bentivogli, Peter Clark, Ido Dagan, Hoa T. Dang, and Danilo Giampiccolo. The seventh PASCAL recognizing textual entailment challenge. 2011.

[11] Anders Björkelund, Love Hafdell, and Pierre Nugues. Multilingual semantic role labeling. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task*, CoNLL '09, pages 43–48, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[12] Bernd Bohnet. Very high accuracy and fast dependency parsing is not a contradiction. In *Proceedings of the 23rd International Conference on Computational Linguistics*, COLING '10, pages 89–97, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[13] Adriane Boyd, Detmar Meurers, Universitaet Tuebingen, and Markus Dickinson. On detecting errors in dependency treebanks.

[14] C. G. Broyden. The convergence of a class of double-rank minimization algorithms: 2. the new algorithm. *IMA J Appl Math*, 6(3):222–231, September 1970.

[15] Sabine Buchholz and Erwin Marsi. CoNLL-X shared task on multilingual dependency parsing. In *In Proc. of CoNLL*, pages 149–164, 2006.

[16] R. C. Bunescu and Raymond J. Mooney. A shortest path dependency kernel for relation extraction. In *Proceedings of the Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP-05)*, pages 724–731, Vancouver, BC, October 2005.

[17] Ekaterina Buyko and Udo Hahn. Evaluating the impact of alternative dependency graph encodings on solving event extraction tasks.

[18] John Carroll, Ted Briscoe, and Antonio Sanfilippo. Parser evaluation: a survey and a new proposal, 1998.

[19] Marie catherine De Marneffe, Bill Maccartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *In LREC 2006*, 2006.

[20] Lin Chang, Chih-Chung and Chih-Jen. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[21] Jinho D. Choi and Martha Palmer. Getting the most out of transition-based dependency parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers - Volume 2*, HLT '11, pages 687–692, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[22] Jinho D. Choi and Martha Palmer. Transition-based semantic role labeling using predicate argument clustering. In *Proceedings of the ACL 2011 Workshop on Relational Models of Semantics*, RELMS '11, pages 37–45, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[23] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14, 1965.

[24] Pere R. Comas, Jordi Turmo, and Luis Marquez. Using dependency parsing and machine learning for factoid question answering on spoken documents. In *Proceedings of the 13th International Conference on Spoken Language Processing (INTERSPEECH 2010)*, Makuhari, Japan, September 2010.

[25] Michael A. Covington. A fundamental algorithm for dependency parsing. In *In Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102, 2000.

[26] Walter Daelemans, Jakub Zavrel, Ko van der Sloot, and Antal van den Bosch. TiMBL: Tilburg memory-based learner - version 4.3 - reference guide, 2002.

[27] J. N. Darroch and D. Ratcliff. Generalized Iterative Scaling for Log-Linear Models. *The Annals of Mathematical Statistics*, 43(5):1470–1480, 1972.

[28] Marie-Catherine de Marneffe and Christopher D. Manning. *Stanford typed dependencies manual*, 2008.

[29] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. Mapreduce: simplified data processing on large clusters. In *In OSDI 04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*. USENIX Association, 2004.

[30] Markus Dickinson and Detmar Meurers. Detecting inconsistencies in treebanks. In *In Proceedings of TLT 2003*, 2003.

[31] Markus Dickinson and W. Detmar Meurers. Prune diseased branches to get healthy trees! how to find erroneous local trees in a treebank and why it matters. In *Proceedings of the Fourth Workshop on Treebanks and Linguistic Theories (TLT 2005)*, Barcelona, Spain, 2005.

[32] Mark Dredze, John Blitzer, Partha P. Talukdar, Kuzman Ganchev, Joao Graca, and Fernando Pereira. Frustratingly Hard Domain Adaptation for Dependency Parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, 2007.

[33] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[34] Leonidas Georgiadis. Arborescence optimization problems solvable by edmonds' algorithm. *Theor. Comput. Sci.*, 301(1-3):427–437, May 2003.

[35] Jesús Giménez and Lluís Màrquez. Svmtool: A general pos tagger generator based on support vector machines. In *Proceedings of the 4th LREC*, Lisbon, Portugal, 2004.

[36] Keith Hall and VÃ¡clav NovÃ¡k. Corrective dependency parsing.

[37] Katri Haverinen, Timo Viljanen, Veronika Laippala, Samuel Kohonen, Filip Ginter, and Tapio Salakoski. Treebanking finnish. In Markus Dickinson, Kaili MÃŒÃŒrisep, and Marco Passarotti, editors, *Proceedings of the Ninth International Workshop on Treebanks and Linguistic Theories (TLT9)*, pages 79–90, 2010.

[38] David Hays. Dependency theory: A formalism and some observations. *Language*, 40:511–525, 1964.

[39] Gann Bierner Jason Baldridge, Tom Morton and Eric Friedman. OpenNLP MaxEnt.

[40] Ronald M. Kaplan and Joan Bresnan. Lexical-functional grammar: A formal system for grammatical representation, 1995.

[41] S. Sathiya Keerthi, Kai-Wei Chang, and et al. A sequential dual method for large scale multi-class linear svms, 2008.

[42] S. Sathiya Keerthi, S. Sundararajan, Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. A sequential dual method for large scale multi-class linear SVMs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 408–416, New York, NY, USA, 2008. ACM.

[43] Dan Klein and Chris Manning. Maxent Models, Conditional Estimation, and Optimization. HLT-NAACL 2003 Tutorial, 2003.

[44] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *IN PROCEEDINGS OF THE 41ST ANNUAL MEETING OF THE ASSOCIATION FOR COMPUTATIONAL LINGUISTICS*, pages 423–430, 2003.

[45] Sandra Kübler, Rachael Cantrell, and Matthias Scheutz. Actions speak louder than words: Evaluating parsers in the context of natural language understanding systems for human-robot interaction. In *Proceedings of the International Conference Recent Advances in Natural Language Processing 2011*, pages 56–62, Hissar, Bulgaria, September 2011. RANLP 2011 Organising Committee.

[46] Jia Li. Linear methods for classification.

[47] Zhenghua Li, Wanxiang Che, and Ting Liu. Improving chinese pos tagging with dependency parsing. In *Proceedings of the 5th International Joint Conference on Natural Language Processing*, 2011.

[48] Dekang Lin. A dependency-based method for evaluating broad-coverage parsers. In *In Proceedings of IJCAI-95*, pages 1420–1425, 1995.

[49] Dekang Lin. Dependency-based evaluation of minipar. In *Proc. Workshop on the Evaluation of Parsing Systems*, Granada, 1998.

[50] Robert Malouf. A comparison of algorithms for maximum entropy parameter estimation. In *Conference on Computational Natural Language Learning (CoNLL)*. Association for Computational Linguistics, 2002.

[51] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2002.

[52] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[53] Ryan Mcdonald, Fernando Pereira, Kiril Ribarov, and Jan Hajic. Non-projective dependency parsing using spanning tree algorithms. In *In Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 523–530, 2005.

[54] Igor Mel'cuk. *Dependency Syntax: Theory and Practice*. State University of New York Press, 1988.

[55] Adam Meyers, Ruth Reeves, Catherine Macleod, Rachel Szekely, Veronika Zielinska, Brian Young, and Ralph Grishman. The NomBank Project: An Interim Report. In Adam Meyers, editor, *HLT-NAACL 2004 Workshop: Frontiers in Corpus Annotation*, pages 24–31, Boston, Massachusetts, USA, May 2004. Association for Computational Linguistics.

[56] Seyed A. Mirroshandel and Alexis Nasr. Active Learning for Dependency Parsing Using Partially Annotated Sentences. In *Proceedings of the 12th International Conference on Parsing Technologies*, pages 140–149, Dublin, Ireland, October 2011. Association for Computational Linguistics.

[57] Yusuke Miyao, Rune Saetre, Kenji Sagae, Takuya Matsuzaki, and Jun'ichi Tsujii. Task-oriented Evaluation of Syntactic Parsers and Their Representations. In *Proceedings of ACL-08: HLT*, pages 46–54, Columbus, Ohio, June 2008. Association for Computational Linguistics.

[58] Yusuke Miyao, Kenji Sagae, Rune Saetre, Takuya Matsuzaki, and Jun ichi Tsujii. Evaluating contributions of natural language parsers to protein-protein interaction extraction. *Bioinformatics*, 25(3):394–400, 2009.

[59] Diego Molla and Ben Hutchinson. Intrinsic versus extrinsic evaluations of parsing systems. In *In Proceedings of EACL Workshop on Evaluation Initiatives in Natural Language Processing*, pages 43–50, 2003.

[60] Joakim Nivre. Dependency Grammar and Dependency Parsing. Technical report, Växjö University: School of Mathematics and Systems Engineering, 2005.

[61] Joakim Nivre. *Inductive Dependency Parsing (Text, Speech and Language Technology)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[62] Joakim Nivre. Algorithms for deterministic incremental dependency parsing. *Comput. Linguist.*, 34(4):513–553, December 2008.

[63] Joakim Nivre. Current trends in data-driven dependency parsing, 2009.

[64] Joakim Nivre. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting ofthe ACL and the 4th International Joint Conference on Natural LanguageProcessing*, pages 351–359, Association for Computational Linguistics Morristown, NJ, USA, 2009. Association for Computational Linguistics Morristown, NJ, USA. Uppsala University.

[65] Joakim Nivre, Johan Hall, and Jens Nilsson. Maltparser: A data-driven parser-generator for dependency parsing. In *In Proc. of LREC-2006*, pages 2216–2219, 2006.

[66] Joakim Nivre and Ryan McDonald. Integrating graph-based and transition-based dependency parsers. In *Proceedings of ACL-08: HLT*, pages 950–958, Columbus, Ohio, June 2008. Association for Computational Linguistics.

[67] Joakim Nivre and Ryan Mcdonald. Integrating graphbased and transition-based dependency parsers. In *In Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-08: HLT*, pages 950–958, 2008.

[68] Joakim Nivre and Jens Nilsson. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 99–106, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[69] Joakim Nivre, Laura Rimell, Ryan McDonald, and Carlos Gomez-Rodriguez. Evaluation of dependency parsers on unbounded dependencies. In *Proceedings of the 23rd International Conference on Computational Linguistics*, COLING '10, pages 833–841, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[70] Partha Pakray, Sivaji Bandyopadhyay, and Alexander Gelbukh. Textual entailment using lexical and syntactic similarity. *International Journal of Artificial Intelligence & Applications (IJAIA), Vol.2, No.1, January 2011*, 2011.

[71] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: An annotated corpus of semantic roles. *Comput. Linguist.*, 31(1):71–106, March 2005.

[72] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. The University of Chicago Press, Chicago, 1994.

[73] Chris Quirk and et al. The impact of parse quality on syntactically-informed statistical machine translation, 2006.

[74] Martin Volk Rico Sennrich, Gerold Schneider and Martin Warin. A new hybrid dependency parser. GSCL-Conference, Potsdam, 2009.

[75] Kenji Sagae. Dependency parsing and domain adaptation with lr models and parser ensembles. In *In Proceedings of the Eleventh Conference on Computational Natural Language Learning*, 2007.

[76] Mihai Surdeanu, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. The conll-2008 shared task on joint parsing of syntactic and semantic dependencies. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, CoNLL '08, pages 159–177, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[77] Mihai Surdeanu and Christopher D. Manning. Ensemble models for dependency parsing: Cheap and good? In *Proceedings of the North American Chapter of the Association for Computational Linguistics Conference (NAACL-2010)*, Los Angeles, CA, June 2010.

[78] Tesniere. *Elements de syntaxe structurale*. Editions Klincksieck, 1959.

[79] Hans van Halteren. The detection of inconsistency in manually tagged text. In *LINC-00*, Luxembourg, 2000.

[80] Alexander Volokh, GÃŒnter Neumann, and Bogdan Sacaleanu. Combining deterministic dependency parsing and linear classification for robust rte. In *Third Text Analysis Conference*. NIST, 2010.

[81] Alexander Volokh and Günter Neumann. 372: Comparing the benefit of different dependency parsers for textual entailment using syntactic constraints only. In *Proceedings of the 5th International Workshop on Semantic Evaluation*, SemEval '10, pages 308–312, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[82] Alexander Volokh and Günter Neumann. Automatic detection and correction of errors in dependency tree-banks. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers - Volume 2*, HLT '11, pages 346–350, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[83] Peng Xu, Jaeho Kang, Michael Ringgaard, and Franz Och. Using a dependency parser to improve smt for subject-object-verb languages. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '09, pages 245–253, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[84] Gisle Ytrestøl. Optimistic backtracking: a backtracking overlay for deterministic incremental parsing. In *Proceedings of the ACL 2011 Student Session*, HLT-SS '11, pages 58–63, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[85] Guo-Xun Yuan, Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. A comparison of optimization methods and software for large-scale l1-regularized linear classification. *J. Mach. Learn. Res.*, 9999:3183–3234, December 2010.

[86] Deniz Yuret. Next generation parser evaluation.

[87] Deniz Yuret, Aydin Han, and Zehra Turgut. Semeval-2010 task 12: Parser evaluation using textual entailments. In *Proceedings of the 5th International Workshop on Semantic Evaluation*, SemEval '10, pages 51–56, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[88] Yi Zhang and Rui Wang. Cross-domain dependency parsing using a deep linguistic grammar.

[89] Yue Zhang and Stephen Clark. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 562–571, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.