Recompiling a knowledge-based dependency parser into memory

Sander Canisius and Antal van den Bosch
ILK / Dept. of Communication and Information Sciences
Tilburg University
P.O. Box 90153, NL-5000 LE Tilburg, The Netherlands
{*S.V.M.Canisius,Antal.vdnBosch*}*@uvt.nl*

Abstract

Data-driven parsers tend to be trained on manually annotated treebanks. In this paper we describe two memory-based dependency parsers trained on treebanks that are automatically parsed by a knowledge-based parser for Dutch. When compared to training on a manual treebank of Dutch, the memory-based parsers exhibit virtually the same performance at the same amount of training material, and achieve markedly higher parsing accuracies when trained on more data. The first memory-based parser is based on a single classifier and operates in linear time, while the second parser employs constraint satisfaction inference (CSI) over three classifiers that each perform a parsing subtask. The non-linear CSI-based parser outperforms the linear parser. Based on this case study we discuss the possibilities of re-engineering knowledge-based parsers in memory.

Keywords

Dependency parsing, memory-based learning, constraint satisfaction inference

1 Introduction

Within the last half century many computational natural language parsers have been designed and implemented. Until a decade ago, most available parsers were rule-based and manually built, drawing on explicitized linguistic knowledge. For instance, for Dutch a prime example is the Alpino parser [9], implementing a HPSG-based stochastic attribute-value grammar. Probably the best parser for Dutch, Alpino is a typical modern example of a rule-based approach that has hybridized with a stochastic, data-driven approach. After a rule-based core generates possible parses for a given sentence (possibly hundreds or thousands), a stochastic component searches in this space of possibilities for the most likely parse, given a background collection of example parses, a so-called *treebank*. Using machine learning methods such as maximum entropy, this stochastic component can be efficiently trained and run [10, 9]. Alpino, available as an open source software system[1], comes with both the parser and the manually annotated treebank on which it was optimized.

The Alpino treebank took several person years to annotate [13], and can now also be used to train any machine-learning-based or stochastic parser on. Nevertheless it has a limited size of about 262 thousand words (currently). Due to the distributional properties of words, sentence-level natural language processing systems based on machine learning tend to improve generalization performance when more data is available [1, 11]. Hence, it is relevant to investigate methods beyond manual annotation by which more annotated data can be harvested and employed.

In the case of Alpino, one route that has not been explored earlier is to take the already existing parser and apply it to a large amount of digitally available unannotated Dutch text, and use the automatically parsed data as (additional) training examples. Of course these parses contain all the errors that Alpino makes, and without human inspection it cannot be known where the errors are. Training a supervised machine learning method on this partly erroneous data will lead to a system that may therefore never be better than the parser. At the same time, the trained system may in fact become as accurate as the parser itself in the long run; it may learn to behave exactly as Alpino would.

In this paper we present an attempt at recompiling the Alpino parser into two variants of a memory-based dependency parser, which is not only trained on Alpino treebank data, converted to dependency structures, but also on large amounts of unannotated texts parsed by Alpino. The two memory-based parsers are tested on various types of text, to test their out-of-domain robustness. It is shown that the two memory-based parsers can improve beyond being trained on the manually annotated treebank, when texts parsed by Alpino are used as training data. Furthermore, both parsers are fast; the faster of the two processes at least 1,500 words per second, its processing time being linear in function of the length of the input sequence.

The paper is structured as follows. In Section 2 we introduce the concept of formulating dependency parsing in a classification framework. We briefly describe IGTree, a fast approximation of k-nearest neighbor classification, which is used as the classifier engine. In Section 3 we provide learning curves, error analyses, and measurements of memory usage and speed of

[1] Alpino: `http://www.let.rug.nl/~vannoord/alp/Alpino/`

the two parsers. We discuss our findings and compare them to the original Alpino parser in Section 4.

2 Algorithms

Two memory-based approaches to dependency parsing are described. Subsequently, IGTree is described [6], which is a fast approximation of the k-nearest neighbor classifier.

2.1 Dependency Parsing as Classification

The first parsing algorithm we present is a straightforward interpretation of dependency parsing as a classification task. Pairs of words are classified as to whether they are connected by a dependency, and if so by which relation type. If this classification is performed for each pair of words in a sentence, many words will be classified as modifying more than one head. In a valid dependency structure however, each word is only to modify one head word. To resolve this issue, [4] propose a simple inference scheme for selecting the most-likely head word for a dependent out of a number of conflicting candidate predicting.

Given a token for which the dependency relation is to be predicted, a number of classification cases have been processed, each of them indicating whether and if so how the token is related to one of the other tokens in the sentence. If all classifications are negative, the token is assumed to have no head. If one of the classifications is non-negative, suggesting a dependency relation between this token as a dependent and some other token as a head, this dependency relation is added to the graph. Finally, there is the case in which more than one prediction is non-negative. By definition, at most one of these predictions can be correct; therefore, only one dependency relation should be added to the graph. The candidates are ranked according to the classification confidence of the base classifier that predicted them, and the highest-ranked candidate is selected for insertion into the graph.

The features we used for encoding instances for this classification task correspond to a rather simple description of the head-dependent pair to be classified. For both the potential head and dependent, there are features encoding a 2-1-2 window of words and part-of-speech tags; in addition, there are two spatial features: a relative position feature, encoding whether the dependent is located to the left or to the right of its potential head, and a distance feature that expresses the number of tokens between the dependent and its head.

2.2 Constraint satisfaction inference for dependency structures

Our second parsing algorithm is an adaptation for dependency structures of the constraint satisfaction inference method for sequential output structures proposed by [5]. The technique uses standard classifiers to predict a weighted constraint satisfaction problem, the solution to which is the complete dependency structure. Constraints that are predicted each cover a small part of the complete structure, and overlap between them ensures that global output structure is taken into account, even though the classifiers only make local predictions in isolation of each other.

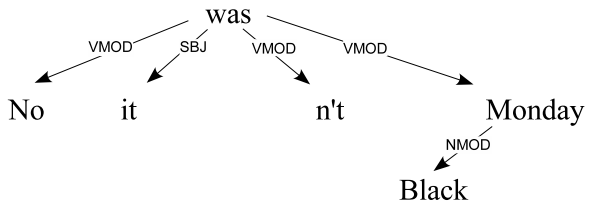

Fig. 1: *Dependency structure for the sentence No it wasn't Black Monday*

A weighted constraint satisfaction problem (W-CSP) is a tuple (X, D, C, W). Here, $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables. $D(x)$ is a function that maps each variable to its domain, and C is a set of constraints on the values assigned to the variables. For a traditional (non-weighted) constraint satisfaction problem, a valid solution is an assignment of values to the variables that (1) are a member of the corresponding variable's domain, and (2) satisfy *all* constraints in the set C. Weighted constraint satisfaction, however, relaxes the requirement to satisfy all constraints. Instead, constraints are assigned weights that may be interpreted as reflecting the importance of satisfying that constraint. The optimal solution to a W-CSP is the solution that assigns those values that maximize the sum of the weights of satisfied constraints.

To adapt this framework to predicting a dependency structure for a sentence, we construct a constraint satisfaction problem by first introducing one variable x_i for each token of the sentence. This variable's value corresponds to the dependency relation that token is the modifier of, i.e. it should specify a relation type and a head token. The constraints of the CSP are predicted by a classifier, where the weight for a constraint corresponds to the classifier's confidence estimate for the prediction.

For the current study, we trained three classifiers to predict three different types of constraints.

1. $C_{dep}(head, modifier, relation)$, i.e. the resulting dependency structure should have a dependency arc from *head* to *modifier* labeled with type *relation*. For the example structure in Figure 1, among others the constraint $C_{dep}(head = was, modifier = No, relation = VMOD)$ should be predicted.

2. $C_{dir}(modifier, direction)$, the relative position (i.e. to its left or to its right) of the head of *modifier*. The structure in Figure 1 will give rise to constraints such as $C_{dir}(modifier = Black, direction = RIGHT)$.

3. $C_{mod}(head, relation)$, in the dependency structure, *head* should be modified by a relation of type *relation*. The constraints generated for the word *was* in Figure 1 would be $C_{mod}(head =$

$was, relations = SBJ$), and $C_{mod}(head = was, relations = VMOD)$.

Predicting constraints of type C_{dep} is essentially what is done by [4]; a classifier is trained to predict a relation label, or a symbol signaling the absence of a relation, for each pair of tokens in a sentence[2]. The training data for this classifier consists of positive examples of constraints to generate, e.g. $was, No, VMOD$, and negative examples, of constraints *not* to generate, e.g. $was, Black, NONE$, but also $No, was, NONE$. In the aforementioned paper, it is shown that down-sampling the negative class in the classifier's training data improves the recall for predicted constraints. The fact that improved recall comes at the cost of a reduced precision is compensated for by our choice for the weighted constraint satisfaction framework: an overpredicted constraint may still be left unsatisfied if other, conflicting constraints outweigh its own weight.

In addition to giving rise to a set of constraints, this classifier differs from the other two in the sense that it is also used to predict the domains of the variables, i.e. any dependency relation not predicted by this classifier will not be considered for inclusion in the output structure.

Whereas the C_{dep} classifier classifies instances for each pair of words, the classifiers for C_{dir} and C_{mod} only classify individual tokens. The features for these classifiers have been kept simple and the same for both classifiers: a 5-slot wide window of both tokens and part-of-speech tags, centered on the token currently being classified. The two classifiers differ in the classes they predict. For C_{dir}, there are only three possible classes: `LEFT`, `RIGHT`, `NONE`. Instances classified as `LEFT`, or `RIGHT` give rise to constraints, whereas `NONE` implies that no C_{dep} constraint is added for that token.

For C_{mod} there is a rather large class space; a class label reflects all modifying relations for the token, e.g. `SBJ+VMOD`. From this label, as many constraints are generated as there are different relation types in the label.

With the above, a weighted constraint satisfaction problem can be formulated that, when solved, describes a dependency structure. As we formulated our problem as a constraint satisfaction problem, any off-the-shelf W-CSP solver could be used to obtain the best dependency parse. However, in general such solvers have a time complexity exponential in the number of variables, and thus in the length of the sentence. As a more efficient alternative we chose to use the CKY algorithm for dependency parsing [7] for computing the best solution, which has only cubic time complexity, but comes with the disadvantage of only considering projective trees as candidate solutions.

Of the two parsing algorithms, the simple classification approach can be expected to be faster than the CSI-based parser that uses the CKY algorithm, and can be expected to be leaner in memory usage due to the fact that it only assumes one classifier as compared to the three classifiers required by the CSI parser. On the other hand, the extra effort spent by the CSI parser is expected to pay off in superior parsing quality. By evaluating both parsers, we attempt to visualize the trade-off between low memory usage and parsing speed on the one hand an parsing quality on the other hand.

[2] For reasons of efficiency and to avoid having too many negative instances in the training data, we follow the approach of [4] of limiting the maximum distance between a potential head and modifier.

2.3 IGTree: A fast approximation of k-NN classification

The classifier engine used in the above-mentioned three classifiers, C_{dep}, C_{dir}, and C_{mod}, is IGTree [6], an algorithm for the top-down induction of decision trees. IGTree compresses a database of labeled examples into a lossless decision-tree structure that preserves the labeling information of all examples, and technically should be named a *trie* according to [8]. A labeled example is a feature-value vector encoding input (in our case, windowed subsequences of words and part-of-speech tags) and output; in our case, labels encoding dependency relations with C_{dep}, the direction of the head of the focus modifier with C_{dir}, and the modifier relations of a focus head with C_{mod}.

An IGTree is a hierarchical tree composed of nodes that each represent a partition of the original example database, and are labeled by the most frequent class of that partition. Besides a majority class label, the nodes also hold complete counts of all class labels in the database partition represented by the node. End nodes (leafs) represent a *homogeneous* partition of the database in which all examples have the same class label. A node is either a leaf or a non-ending node, which branch out to nodes at deeper levels of the trie. Each branch represents a test on a feature value; branches fanning out of one node test on values of the same feature.

Classification in IGTree occurs according to standard decision-tree classification; a new example (i.e. an unlabeled feature vector) is matched deterministically, top-down, against paths in the tree, until an end node is met, or no branch in the tree matches with the value at the particular feature tested at that node; the class label of that last visited node or end node is the classifier's prediction. As a normalized measure of confidence for the predictions made by IGTree, needed in the inference step of the dependency parser, we divide the tree-node counts assigned to the majority class found at the last visited node, by the total counts assigned to all classes at that node. Though this confidence measure is a rather ad-hoc one, which should certainly not be confused with any kind of probability, it tends to work quite well in practice [4].

3 Performance analyses

In this section we make several observations on the performance of the various memory-based dependency parsers. We test their generalization accuracies with learning curves, and apply them to several test sets to test their coverage and robustness. We briefly analyze the most frequent errors of the best parser (the CSI-based parser trained on the concatenation of all available material), and end the section with an analysis of memory usage and speed.

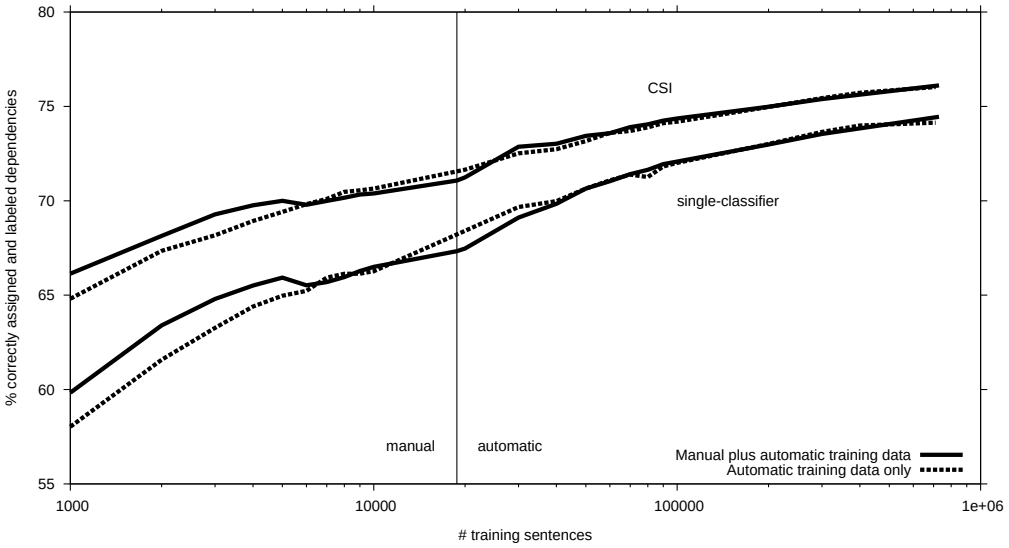

Fig. 2: *Dependency parsing learning curves in terms of the percentage of correctly labeled dependencies, trained on manual plus automatic data (solid line), or only on automatically parsed training data (dashed line), both of the C_{dep} parser ("single-classifier", bottom two lines) and of the CSI-based parser combining C_{dep}, C_{dir}, and C_{mod} ("CSI", upper two lines).*

3.1 Generalization performance and coverage

As training material for the two memory-based dependency parsers we used all manually annotated data available in the Alpino Treebank [13], amounting to 18,791 sentences with 262,452 words. The treebank follows the Spoken Dutch Corpus treebank format, which is to a limited degree constituent-based, but which contains all necessary information to convert each tree to a dependency structure. We did this using the conversion software employed in the CoNLL-X shared task [3]. To generate training material for C_{dep}, the pair-wise dependency relations task, we converted all dependency graphs to 2,959,456 pairwise examples, and subsequently down-sampled the negative class to a relative occurrence compared to non-negative examples of 2 : 1 [4] to 726,440 examples. For the two additional classification tasks of the CSI-based parser, C_{dir} and C_{mod}, we generated training sets of 262,452 examples (i.e. the number of words in the treebank).

Subsequently, texts were collected that were automatically parsed by the Alpino parser [9]: ten thousand Dutch Wikipedia pages (about 179 thousand sentences, 2.2 million words), newspaper articles from the *Algemeen Dagblad* from the first half of 1994 (about 498 thousand sentences, 8.1 million words), and the unannotated parts of the *Eindhoven corpus* (33 thousand sentences, 551 thousand words). Instead of the part-of-speech tags furnished by Alpino, we re-tagged the corpus with the rich Spoken Dutch Corpus tagset, using a fast memory-based tagger [12]. This large retagged corpus of approximately 710 thousand sentences and 10.8 million words is subsequently converted and down-sampled to 28.8 million pairwise C_{dep} examples. Also, 10.8 million examples of both the C_{dir} and C_{mod} classifier were generated.

With these various labeled training sets, three variants of the two memory-based dependency parsers are trained. The first variant of both parsers is trained only on the manually annotated data; the second is trained exclusively on the automatically annotated data, while the third is trained on a concatenation of both training sets. Figure 2 displays learning curves in terms of correctly assigned and labeled dependencies, a commonly used evaluation metric [3], of the three variants, for both parsers. The x axes of the figure has a logarithmic scale and represents the number of training sentences. Two curves are plotted per parser rather than three, as the learning curve of the concatenated set continues at the point where the manual training set stops (i.e. at 18,791 sentences, indicated by the vertical bar) – the learning curve of the manual training set is subsumed by the curve of the concatenated set.

The test set consists of 2,530 sentences (47,471 words) taken from the manually parsed section of the Eindhoven corpus (the *cdbl* part) that is held out from the training data; this is professionally written newspaper text with relatively long sentences, with many subclauses and quotations.

For each parser, the two curves are remarkably similar; training a parser on automatically parsed training data leads to virtually the same accuracies as training on manually annotated data. Also, continuing training a parser on automatically parsed data does not cause the learning curve to regress.

As can be clearly observed from the learning curves, the CSI-based parser performs consistently better than the single-classifier parser, but with a diminishing gap as more training examples are available. At the current maximal amount of training data, approximately 729 thousand sentences, the difference is about 1.6%.

The best scores of the two parsers trained exclusively on three variants of different kinds of training data, and tested on the aforementioned manually

Parser	Evaluation	Newspaper text Manual	Automatic	Both	Questions Automatic	Test suite Automatic
Single-classifier	Labeled dependencies	67.3	74.1	74.5	78.7	77.0
	Unlabeled dependencies	70.6	76.9	77.2	82.4	78.8
	Label accuracy	76.3	81.2	81.4	81.6	83.3
CSI-based	Labeled dependencies	71.1	76.0	76.1	80.6	79.9
	Unlabeled dependencies	75.2	79.3	79.4	84.5	82.5
	Label accuracy	77.2	81.2	81.2	82.7	83.6

Table 1: *Best accuracies on test data of the linear and* CSI*-based parser: the percentage of correctly assigned dependencies, with and without labeling, and the accuracy on labels only, tested on newspaper texts, a test set of questions, and a test suite of 18 hand-selected sentences.*

parsed test set, are displayed in Table 1. The table also includes accuracy scores on correctly assigned dependency relations regardless of the label ("unlabeled dependencies"), and on correctly assigned labels regardless of which word the word relates with ("label accuracy"). The parser trained exclusively on automatically parsed data is also tested along the same evaluation metrics on two different test sets that are part of the manually annotated training set, namely a set of 1,100 questions from the CLEF Dutch question-answering competition[3], and a test suite of 18 sentences used in a comparison of Dutch parsers in 2001. The parser trained on automatically parsed data performs at accuracy levels comparable to the scores on the first test set. From this it can be tentatively concluded that the parser indeed has a wide coverage.

3.2 Error analysis of the best parser

The best-performing parser is the CSI-based parser trained on the concatenation of all available manually-annotated and automatically-annotated training data. It achieves a labeled dependency accuracy of 76.11% (36,132 out of 47,471 words), an unlabeled dependency accuracy of 79.39% (37,685 out of 47,471 words), and a label accuracy of 81.24% (38,565 out of 47,471 words).

Analyzed at the level of part-of-speech tags, the most accurately attached word classes are punctuation (99%) and pronouns (96%), while prepositions (57%, especially the words *in*, in, and *op*, on) and conjunctions (55%, especially the word *en*, and) are the hardest to assign a correct dependency to.

The easiest dependency relations are *punctuation* (99% precision and recall) and *determiner* (95% recall, 97% precision). The most frequent relation, *modifier*, is assigned with a precision of 71% and a recall of 67%. The second-most frequent relation, *direct object*, is assigned with a precision of 85% and a recall of 77%. Third, the *subject* relation is assigned with a precision of 74% and a recall of 71%.

The most common error of the parser that could in fact be post-processed and partly corrected, is the assignment of the ROOT relation (generally, the relation between the main verb to the "root" node) one or two times too many.

[3] CLEF: `http://www.clef-campaign.org/`

Training set	Single-classifier	CSI-based
Manually annotated	3.5	8.9
Automatically parsed	33.7	87.6
Both	34.4	89.5

Table 2: *Amount of memory used (Mb) by the single-classifier parser and the CSI-based parser with the two training set sizes and their combination.*

3.3 Memory usage and speed

The memory footprint of the single-classifier C_{dep} parsers and the three-module CSI-based parser, as well as their processing speed is also measured[4]. Table 2 summarizes the memory-usage measurements of the single-classifier parser and the CSI-based parser when trained on maximal amounts of training data. The footprint of the parser trained on manually annotated data is small (under 10 Mb), but this is at the cost of a lower performance. Trained on all available automatically-annotated and manually-annotated training data, the single-classifier parsers have a footprint of about 34 Mb, which can still be regarded reasonable in current computers. Their CSI-based counterparts also have to claim memory for the C_{dir} and C_{mod} classifiers, hence they have a larger memory need of about 89 Mb.

Typically, rule-based parsers become exponentially slower when parsing longer sentences. Alpino uses stochastic search to battle the problem, but the solution is only partial. To get an idea of the behavior of the memory-based approach with longer sentences, the speed and accuracy of both the single-classifier parser and the CSI-based parser was measured on different sentence lengths found in the first test set. Figure 3 shows both, measured on sentence lengths from 2 to 50. As the left graph of Figure 3 shows, shorter sentences are parsed more successfully, which is also typical for Alpino; the CSI-based parser furthermore outperforms the single-classifier consistently. The right graph of Figure 3 shows a perhaps more unexpected leveling of the speed of the single-classifier parser to about 1,500 words per second; sentences shorter than 20 words are processed faster. The roughly linear speed for longer sentences may be unexpected. Ear-

[4] The hardware used for testing is equipped with Dual Core AMD Opteron 880 2,412 Mhz processors.

lier we noted that for each sentence pairwise examples are generated ($n(n-1)$, to be exact), but we also constrained this (also with test sentences) to pairs of words within a range of eight words from each other, as 95% of all relations in the training corpus occur within that range. This fixed constraint bounds the number of examples per sentence, making the relation between the sentence length and the number of examples effectively linear.

The CSI-based parser is slower than the single-classifier parser for two reasons: first, it is based on three classifiers (C_{dep}, C_{dir}, and C_{mod}), rather than the single C_{dep}, which alone makes it roughly five times slower (the C_{dir} and C_{mod} classifiers are each about twice as slow as the C_{dep} classifier). Second, the CSI-based parser performs an extra inference step, the CKY algorithm, to arrive at a full dependency structure. As said, processing time of this algorithm is cubic in the length of the input. Beyond sentence length 10, the CKY procedure takes more time than the three (linearly processing) classifiers. Effectively, the speed appears to diminish at a linear rate, from about 700 words per second for very short sentences, via about 350 words per second at 20 words, the average length of sentences in the test set, to about 170 words per second at sentence length 40. Note that the parser can process sentences of any length; furthermore, both parsers never fail to process a sentence.

4 Discussion

The experiments in this paper have shown that a manually written knowledge-based parser can to some extent be re-engineered as a memory-based parser, which performs similarity-based reasoning on examples of fragments of parses generated by the original parser. Recompilations in memory can be quite fast when using IGTree, a fast approximation of k-nearest neighbor classification, as the classifier engine. We developed a single-classifier parser operating in linear time, which processes sentences at speeds of at least 1,500 words per second. The second, more complex parser based on three classifiers and with a constraint-satisfaction inference step built in is slower; it only processes a few hundred words per second. The longer the sentence, the slower the CSI-based parser. Yet, there is no exponential increase in processing time with very long sentences. Furthermore, both parsers never fail to parse a sentence, and tests on three different test texts showed a robustly consistent level of performance.

A vexing question is whether we have actually built parsers that emulate, or may in the long run emulate, Alpino. It is clear that both parsers can never be pure emulations; they may produce different results on unseen text than Alpino does, if only because of the fact that the stochastic component of Alpino may prefer a different dependency structure over the one that the CSI-based parser produces, which does not operate on the corpus-derived probabilities that Alpino uses.

For now it may be illustrative to compare evaluations on the same test texts. Alpino has been evaluated with various metrics; in [9] the designers of Alpino argue for using an adapted form of *concept accuracy* to estimate the correctness of the dependency labeling. The labeled dependencies accuracy metric of the CoNLL-X shared task [3], based on an earlier proposal by [2], used throughout this paper, does exactly that. Both metrics essentially compute *#correct/#total*, i.e., the number of correctly assigned relations divided by the total number of relations. The only difference between the two metrics is that the Alpino proposal uses the larger one of the number of relations in the target structure versus in the predicted structure as *#total*, while the CoNLL-X labeled dependencies metric simply takes the number of relations in the target structure as the denominator. For our parsers, the number of generated relations will never be larger than the number of tokens, hence the simple labeled dependency accuracy metric suffices.

Test set	Alpino parser concept accuracy	CSI-based parser labeled dependencies
Newspaper	87.9	76.1
Questions	88.7	80.6

Table 3: *Comparison of concept accuracy scores of Alpino, and labeled dependency accuracies of our best parser on similar test texts.*

For Alpino, however, the situation is different, and herein lies the major difference between our parsers and Alpino: the latter adopts the Spoken Dutch Corpus syntactic tree format, which uses non-terminal labels of constituents to a limited degree. For example, in the sentence *I saw Cathy wave wildly* (and analogously in the Dutch translation), our parsers would relate the head *saw* to the verbal complement *wave*, which in turn would be the head in relation to both *Cathy* (the subject of *wave*) and *wildly* (the adverbial modifier of *wave*). In the Alpino treebank, however, the three tokens of *Cathy wave wildly* all relate to a non-terminal *VC* (verbal complement); *wave* is the modifier of a "head verb"-relation with this non-terminal (i.e., it is its head). In turn, the non-terminal is the modifier of a verbal complement relation to the root node of the sentence, with which *saw* has a "head verb" relation. In general this means that an Alpino parse may contain more relations than in the target structure, as the parser may overgenerate non-terminals, explaining the argument put forward by [9] for their concept accuracy metric.

Given this, we cannot directly compare our parsers to Alpino. Still, it is interesting to contrast some results obtained on the same or similar test sets. In Table 3 we contrast scores reported in [9] on a newspaper corpus, similar to our first test set, and scores on a part of the question collection we also tested on (cf. Table 1). Alpino's scores are higher, but as mentioned, no direct comparison can be made.

Besides generalization accuracy, our parsers differ in the amount of memory and in speed with Alpino. Although the Alpino parser is quite memory-lean, it needs more memory with larger sentences. In contrast, our parsers have a static memory footprint (apart from an additional modest cubic-size buffer needed by the CSI-based parser). In terms of speed, Alpino can be exceptionally slow with long sentences due to its ex-

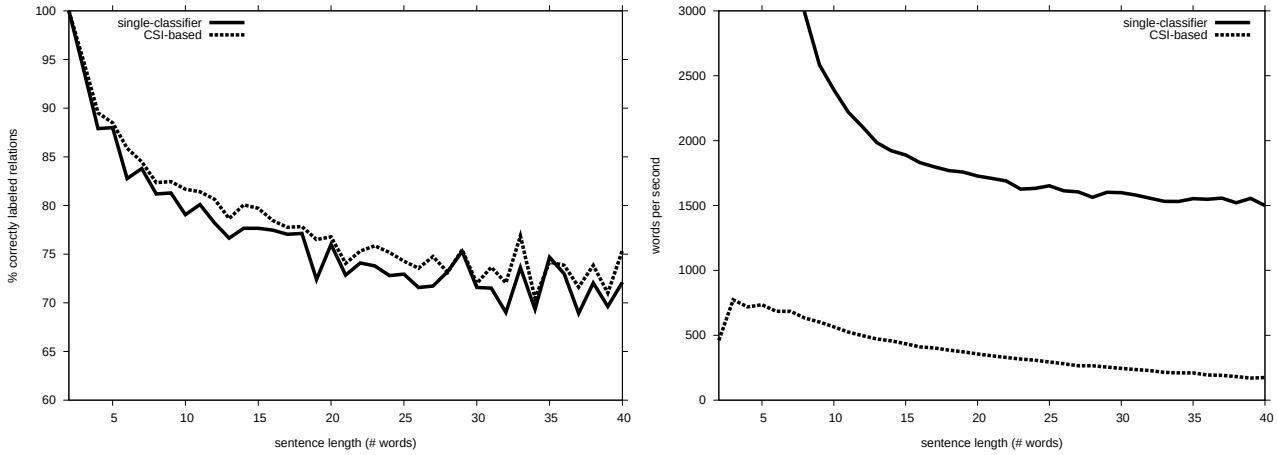

Fig. 3: *Generalization accuracies in terms of percentages of correctly labeled dependencies (left) and words processed per second (right) of the single-classifier and CSI-based dependency parsers trained on the maximal amount of data, measured per sentence length from 2 to 50.*

ponential components, and needs considerable search heuristics and even memory and time limits to keep within reasonable bounds; in contrast, our parsers appear to behave either linear (the single-classifier parser) or only slightly slower (the CSI-based parser) when processing longer sentences.

In future work we intend to compare Alpino directly to our approach by converting Alpino's output to our dependency structures. We plan to add one or more simple classifiers to the current classifier pool of the CSI-based parser, as we have not exhausted yet the possibilities of defining elemental dependency parsing subtasks. Finally, we intend to continue training on more texts parsed by Alpino, as the end of that resource, and of the ensuing learning curve, is not in sight.

Acknowledgments

We are indebted to Gertjan van Noord and his co-workers at the Rijksuniversiteit Groningen, The Netherlands, for their invaluable work on the Alpino parser and treebank. This work was supported by NWO, the Netherlands Organisation for Scientific Research, in the context of the NWO IMIX Programme and the NWO Vici project "Implicit linguistics".

References

[1] M. Banko and E. Brill. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, pages 26–33. Association for Computational Linguistics, 2001.

[2] T. Briscoe, J. Carroll, J. Graham, and A. Copestake. Relational evaluation schemes. In *Proceedings of the Beyond PARSEVAL Workshop at the 3rd International Conference on Language Resources and Evaluation*, pages 4–8, Las Palmas, Gran Canaria, 2002.

[3] S. Buchholz and E. Marsi. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of CoNLL-X, the Tenth Conference on Computational Natural Language Learning*, New York, NY, 2006.

[4] S. Canisius, T. Bogers, A. Van den Bosch, J. Geertzen, and E. Tjong Kim Sang. Dependency parsing by inference over high-recall dependency predictions. In *Proceedings of the Tenth Conference on Computational Natural Language Learning, CoNLL-X*, New York, NY, 2006.

[5] S. Canisius, A. Van den Bosch, and W. Daelemans. Constraint satisfaction inference: Non-probabilistic global inference for sequence labelling. In *Proceedings of the EACL 2006 Workshop on Learning Structured Information in Natural Language Applications*, pages 9–16, Trento, Italy, 2006.

[6] W. Daelemans, A. Van den Bosch, and A. Weijters. IGTree: using trees for compression and classification in lazy learning algorithms. *Artificial Intelligence Review*, 11:407–423, 1997.

[7] J. Eisner. Bilexical grammars and their cubic-time parsing algorithms. *Advances in Probabilistic and Other Parsing Technologies*, pages 29–62, 2000.

[8] D. E. Knuth. *The art of computer programming*, volume 3: Sorting and searching. Addison-Wesley, Reading, MA, 1973.

[9] R. Malouf and G. Van Noord. Wide coverage parsing with stochastic attribute value grammars. In *Proceedings of the IJCNLP-04 Workshop Beyond Shallow Analyses - Formalisms and statistical modeling for deep analyses*, 2004.

[10] M. Osborne. Estimation of stochastic attribute-value grammars using an informative sample. In *Proceedings of the 18th International Conference on Computational Linguistics, COLING-2000*, 2000.

[11] A. Van den Bosch and S. Buchholz. Shallow parsing on the basis of words only: A case study. In *Proceedings of the 40th Meeting of the Association for Computational Linguistics*, pages 433–440, 2002.

[12] A. Van den Bosch, I. Schuurman, and V. Vandeghinste. Transferring PoS-tagging and lemmatization tools from spoken to written Dutch corpus development. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation, LREC-2006*, Trento, Italy, 2006.

[13] L. Van der Beek, G. Bouma, R. Malouf, and G. Van Noord. The alpino dependency treebank. In *Selected Papers from the Twelfth Computational Linguistics in the Netherlands Meeting, CLIN-2001*, Amsterdam, 2001. Rodopi.