Learning Efficient Parsing

Gertjan van Noord
University of Groningen
G.J.M.van.noord@rug.nl

Abstract

A corpus-based technique is described to improve the efficiency of wide-coverage high-accuracy parsers. By keeping track of the derivation steps which lead to the best parse for a very large collection of sentences, the parser learns which parse steps can be filtered without significant loss in parsing accuracy, but with an important increase in parsing efficiency. An interesting characteristic of our approach is that it is self-learning, in the sense that it uses unannotated corpora.

1 Introduction

We consider wide-coverage high-accuracy parsing systems such as Alpino, a parser for Dutch which contains a grammar based on HPSG and a maximum entropy disambiguation component trained on a treebank. Even if such parsing systems now obtain satisfactory accuracy for a variety of text types, a drawback concerns the computational properties of such parsers: they typically require lots of memory and are often very slow for longer and very ambiguous sentences.

We present a very simple, fairly general, corpus-based method to improve upon the practical efficiency of such parsers. We use the accurate, slow, parser to parse many (unannotated) input sentences. For each sentence, we keep track of sequences of derivation steps that were required to find the *best* parse of that sentence (i.e., the parse that obtained the best score, highest probability, according to the parser itself).

Given a large set of successful derivation step sequences, we experimented with a variety of simple heuristics to filter unpromising derivation steps. A heuristic that works remarkably well simply states that for a new input sentence, the parser can only consider derivation step sequences in which any sub-sequence of length N has been observed at least once in the training data. Experimental results are provided for various heuristics and amounts of training data.

It is hard to compare fast, accurate, parsers with slow, slightly more accurate parsers. In section 3 we propose both an on-line and an off-line application scenario, introducing a time-out per sentence, which leads to metrics for choosing between parser variants.

In the experimental part we show that, in an online scenario, the most successful heuristic leads to a parser that is more accurate than the baseline system, except for unrealistic time-outs per sentence of more than 15 minutes. Furthermore, we show that, in an off-line scenario, the most successful heuristic leads to a parser that is more than four times faster than the base-line variant with the same accuracy.

2 Background: the Alpino parser for Dutch

The experiments are performed using the Alpino parser for Dutch. The Alpino system is a linguistically motivated, wide-coverage grammar and parser for Dutch in the tradition of HPSG. It consists of about 800 grammar rules and a large lexicon of over 300,000 lexemes and various rules to recognize special constructs such as named entities, temporal expressions, etc. Heuristics have been implemented to deal with unknown words and word sequences. Based on the categories assigned to words, and the set of grammar rules compiled from the HPSG grammar, a left-corner parser finds the set of all parses, and stores this set compactly in a packed parse forest. In order to select the best parse from the parse forest, a best-first search algorithm is applied. The algorithm consults a Maximum Entropy disambiguation model to judge the quality of (partial) parses.

Although Alpino is not a dependency grammar

in the traditional sense, dependency structures are generated by the lexicon and grammar rules as the value of a dedicated attribute. The dependency structures are based on CGN (Corpus Gesproken Nederlands, Corpus of Spoken Dutch) (Hoekstra et al., 2003), D-Coi and LASSY (van Noord et al., 2006).

3 Methodology: balancing efficiency and accuracy

3.1 On-line and off-line parsing scenarios

We focus on the speed of parsing, ignoring other computational properties such as memory usage. Problems with respect to parsing are twofold: on the one hand, parsing simply is too slow for many input sentences. On the other hand, the relation between input sentence and expected speed of parsing is typically unknown. For simple parsing systems based on finite-state, context-free or mildly context-sensitive grammars, it is possible to establish an upper-bound of required CPU-time based on the length of an input sentence. For the very powerful constraint-based formalisms considered here, such upper-bounds are not available. In practice, shorter sentences typically can be parsed fairly quickly, whereas longer sentences sometimes can take a very very long time indeed. As a consequence, measures such as number of words parsed per minute, or mean parsing time per sentence are somewhat meaningless. We therefore introduce two slightly different scenarios which include a time-out per sentence.

On-line scenario. In some applications, a parser is applied *on-line*: an actual user is waiting for the response of the system, and if the parser required minutes of CPU-time, the application would not be successful. In such a scenario, we assume that it is possible to determine a maximum amount of CPU-time (a time-out) per sentence, depending on other factors such as speed of the other system components, expected patience of users, etc. If the parser does not finish before the time-out, it is assumed to have not produced anything. In dependency parsing, the parser produces the empty set of dependencies in such cases, and hence such an event has an important negative effect on the accuracy of the system. By studying the relation between different time-outs and accuracy, it is possible to choose the most effective parser variant for a particular application.

Off-line scenario. For other applications, an off-line parsing scenario might be more appropriate. For instance, if we build a question answering system for a medical encyclopedia, and we wish to parse all sentences of that encyclopedia once and for all, then we are not interested in the amount of CPU-time the parser spends on a single sentence, but we want to know how much time it will cost to parse everything.

In such a scenario, it often still is very useful to set a time-out for each sentence, but in this case the time-out can be expected to be (much) higher than in the on-line scenario. In this scenario, we propose to study the relation between mean CPU-time and accuracy – for various settings of the time-out parameter. This allows us to determine, for instance, the mean CPU-time requirements for a given target accuracy level?

3.2 Accuracy: comparing sets of dependencies

Let D_p^i be the number of dependencies produced by the parser for sentence i, D_g^i is the number of dependencies in the treebank parse, and D_o^i is the number of correct dependencies produced by the parser. If no superscript is used, we aggregate over all sentences of the test set, i.e.,:

$$D_p = \sum_i D_p^i \quad D_o = \sum_i D_o^i \quad D_g = \sum_i D_g^i$$

We define precision ($P = D_o/D_p$), ($R = D_o/D_g$) and f-score: $2\mathrm{P} \cdot \mathrm{R}/(\mathrm{P} + \mathrm{R})$.

An alternative similarity score is based on the observation that for a given sentence of n words, a parser would be expected to return (about) n dependencies. In such cases, we can simply use the percentage of correct dependencies as a measure of accuracy. To allow for some discrepancies between the number of expected and returned dependencies, we divide by the maximum (per sentence) of both. This leads to the following definition of *named dependency accuracy*.

$$\mathrm{Acc} = \frac{D_o}{\sum_i \max(D_g^i, D_p^i)}$$

If time-outs are introduced, the difference between f-score and accuracy becomes important. Consider the example in table 1. Here, the parser produces reasonable results for the first three, short, sentences, but for the final, long, sentence no result is produced because of a time-out.

i	D_o^i	D_p^i	D_g^i	prec	rec	f-sc	Acc
1	8	10	11	80	73	76	73
2	8	11	10	76	76	76	73
3	8	9	9	80	80	80	77
4	0	0	30	80	40	53	39

Table 1: Hypothetical result of parser on a test set of four sentences. The columns labeled precision, recall, f-score and accuracy represent aggregates over sentences $1 \ldots i$.

The precision, recall and f-score after the first three sentences is 80%. After the – much longer – fourth sentence, recall drops considerably, but precision remains the same. As a consequence, the f-score is quite a bit higher than 40%: it is over 53%. The accuracy score after three sentences is 77%. Including the fourth sentence leads to a drop in accuracy to 39%.

As this example illustrates, the f-score metric is less sensitive to parse failures than the accuracy score. Also, it appears that the accuracy score is a much better characterization of the success of this parser: after all, the parser only got 24 correct dependencies out of 60 expected dependencies. The f-score measure, on the other hand, can easily be misunderstood to suggest that the parser does a good job for more than 50%.

4 Learning Efficient Parsing

In this section a method is defined for filtering derivation step sequences, based on previous experience of the parser. In a training phase, the parser is fed with thousands of sentences. For each sentence it finds the best parse, and it stores the relevant sequences of derivation steps, that were required to find that best parse. After the training phase, the parser filters those sequences of derivation steps that are unlikely to be useful. By filtering out unlikely derivation step sequences, efficiency is expected to improve. Since certain parses now become impossible, a drop in accuracy is expected as well.

Although the idea of filtering derivation step sequences based on previous experience is fairly general, we define the method in more detail with respect to an actual parsing algorithm: the left-corner parser along the lines of Matsumoto et al. (1983), Pereira and Shieber (1987, section 6.5) and van Noord (1997).

4.1 Left-corner parsing

A left-corner parser is a bottom-up parser with top-down guidance, which is most easily explained as a non-deterministic search procedure. A specification of the left-corner algorithm can be provided in DCG as in figure 2 (Pereira and Shieber, 1987, section 6.5), where the `filter/2` goals should be ignored for the moment. Here, we assume that dictionary look-up is performed by the `word/3` predicate, with the first argument a given word, and the second argument its category; and that rules are accessible via the predicate `rule/3`, where the first argument represents the mother category, and the second argument is the possibly empty list of daughter categories. The third argument of both the `word/3` and `rule/3` predicates are identifiers we need later.

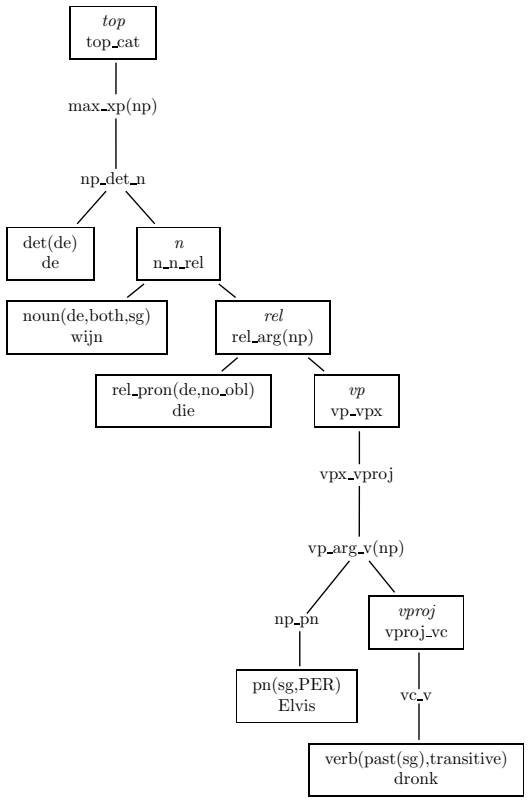
In order to analyze a given sentence as an instance of the top category, we look up the first word of the string, and show that this lexical category is a *left-corner* of the *goal category*. To show that a given category is a left-corner of a given *goal category*, a rule is selected. The leftmost daughter node of that rule is identified with the left-corner. The other daughters of the rule are parsed recursively. If this succeeds, it remains to show that the mother node of the rule is a left-corner of the *goal category*. The recursion stops if a left-corner category can be identified with the goal category.

This simple algorithm is improved and extended in a variety of ways, as in Matsumoto et al. (1983) and van Noord (1997), to make it efficient and practical. The extensions include a memoization of the *parse/1* predicate and the construction of a shared parse forest (a compact representation of all parses).

4.2 Left-corner splines

For the left-corner parser, the derivation step sequences that are of interested are left-corner splines. Such a spline consists of a *goal category*, and the rules and lexical entries which were used in the left-corner, in the order from the top to the bottom.

A spline consists of a goal category, followed by a sequence of derivation step names. A derivation step name is typically a rule identifier, but it can also be a lexical type, indicating the lexical category of a word that is the left-corner. A special derivation step name is the reserved symbol




```
(top,[finish,top_cat,max_xp(np),np_det_n,det(de)]).
(n,[finish,n_n_rel,noun(de,both,sg)]).
(rel,[finish,rel_arg(np),rel_pron(de,no_obl)]).
(vp,[finish,vp_vpx,vpx_vproj,vp_arg_v(np),np_pn,pn(sg,PER),]).
(vproj,[finish,vproj_vc,vc_v,verb(past(sg),transitive)]).
```

Figure 1: Annotated derivation tree of the sentence *De wijn die Elvis dronk* (The wine which Elvis drank).

`finish` which is used to indicate that the current category is identified with the goal category (and no further rules are applied). A spline is written $(g, r_n \ldots r_1)$ for goal category g and derivation step names $r_1 \ldots r_n$. $(g, r_i \ldots r_1)$ is a partial spline of $(g, r_n \ldots r_i \ldots r_1)$.

Consider the annotated derivation tree for the sentence *De wijn die Elvis dronk* (The wine which Elvis drank) in figure 1. Boxed leaf nodes contain the lexical category as well as the corresponding word. Boxed non-leaf nodes contain the goal category (italic) and the rule-name. Non-boxed non-leaf nodes only list the rule name. The first left-corner spline consists of the goal category `top` and the identifiers `finish`, `top_cat`, `max_xp(np)`, `np_det_n`, and the lexical type `det(de)`. All five left-corner splines of the example are listed at the bottom of figure 1.

Left-corner splines of best parses of a large set of sentences constitute the training data for the techniques we develop to learn to parse new sentences more efficiently.

```
parse(Phrase) -->
    leaf(SubPhrase,Id),
    { filter(Phrase,[Id]) },
    lc(SubPhrase,Phrase,[Id]).

leaf(Cat,Id) -->
    [Word], { word(Word,Cat,Id) }.
leaf(Cat,Id) --> { rule(Cat,[],Id) }.

lc(Phrase,Phrase,Spline) -->
    { filter(Phrase,[finish|Spline]) }.
lc(SubPhrase,SuperPhrase,Spline) -->
    rule(Phrase,[SubPhrase|Rest],Id),
    { filter(SuperPhrase,[Id|Spline]) },
    parse_rest(Rest),
    lc(Phrase,SuperPhrase,[Id|Spline]).
```

Figure 2: DCG Specification of a non-deterministic left-corner parser, including spline filtering.

4.3 Filtering left-corner splines

The left-corner parser builds left-corner splines one step at the time. For a given goal, it first selects a potential left-corner, and then continues applying rules from the bottom to the top until the left-corner is identified with the goal category. At every step where the algorithm attempts to extend a left-corner spline, we now introduce a filter. The purpose of this filter is to consider only those partial left-corner splines that look promising - based on the parser's previous experience on the training data. The specification of the left-corner parser given in figure 2 includes calls to this filter.

The purpose of the filter is, that at any time the parser considers extending a left-corner spline $(g, r_{i-1} \ldots r_1)$ to $(g, r_i \ldots r_1)$, such an extension only is allowed in promising cases. Obviously, there are many ways such a filter could be defined. We identify the following dimensions:

Context size. A filter for $(g, r_i \ldots r_1)$ will typically ignore at least some of the derivation step names from the context. We experiment with filters which take into consideration g, r_i, r_{i-1} (*bigram filter*); g, r_i, r_{i-1}, r_{i-2} (*trigram filter*); and $g, r_i, r_{i-1}, r_{i-2}, r_{i-3}$ (*fourgram filter*). A further filter, labeled *prefix filter*, takes the full history into account: $g, r_i \ldots r_1$. The prefix filter thus ensures that the parser only considers left-corner splines that are partial splines of splines observed in the training data.

Required evidence. For the various filters, what kind of evidence from the training data do we require in order for the filter to accept this particular derivation step? In initial experiments, we used relative frequencies. For instance, the trigram filter would allow any tuple g, r_{i-2}, r_{i-1}, r_i for some constant threshold τ, provided:

$$\frac{C(g, \ldots r_i r_{i-1} r_{i-2} \ldots)}{C(g, \ldots r_{i-1} r_{i-2} \ldots)} > \tau$$

However, we found that filters are more effective (and require much less space – see below), which simply require that every step has been observed often enough in the training data:

$$C(g, \ldots r_i r_{i-1} r_{i-2} \ldots) > \tau$$

In particular, the case where $\tau = 0$ gave surprisingly good results.

4.4 Comparison with link table

The filter we developed is reminiscent of the *link* predicate of (Pereira and Shieber, 1987). An important difference with the filter developed here is that the *link* predicate removes derivation steps which cannot lead to a successful parse (by an off-line global analysis of the grammar), whereas we filter out derivation steps which *can* lead to a full parse, but which are not expected to lead to a *best* parse. In our implementation, a variant of the *link* predicate is used as well.

4.5 Implementation detail

The definition of the filter predicate depends on our choices with respect to the dimensions identified above. For instance, if we chose the trigram filter as our context size, then the training data can be preprocessed in order to store all goal-trigram-pairs with frequency above the threshold τ. During parsing, if the filter is given the partial spline $(g, r_i r_{i-1} r_{i-2} \ldots)$, then a simple table look-up for the tuple $(g, r_{i-2} r_{i-1} r_i)$ is sufficient (this suffices, because each of the preceding trigrams will have been checked earlier). In general, the filter predicate needs access to a table containing a pair of goal category and context, where the context consists of sequences of derivation step names. The table contains items for those pairs that occurred with frequency $> \tau$ in the training data.

To access such tables efficiently, an obvious choice is to use a hash table. The additional storage requirements for such a hash table are considerable. For instance, for the prefix filter four years of newspaper text lead to a table with 941,723 entries - stored as text the data takes 103Mb. To save space, we experimented with a set-up in which only the hash keys are stored, but the original information that the hash key was computed from, is removed. During parsing, in order to check that a given tuple is allowable, we compute its hash key, and check if the hash key is in the table. If so, the computation continues. The drawback of this method is, that in the case a hash collision would have occurred in an ordinary hash table, we now simply assume that the input tuple was in the table. In other words: the filter is potentially too permissive in such cases. In actual practice, we did not observe a difference with respect to accuracy or CPU-time requirements, but the storage costs dropped considerably.

5 Experimental Results

Some of the experiments have been performed with the Alpino Treebank. The Alpino Treebank (van der Beek et al., 2002) consists of manually verified dependency structures for the `cdbl` (newspaper) part of the Eindhoven corpus (den Boogaart, 1975). The treebank contains 7137 sentences. Average sentence length is about 20 tokens.

Some further experiments are performed on the basis of the D-Coi corpus (van Noord et al., 2006). From this corpus, we used the manually verified syntactic annotations of the P-P-H and P-P-L parts. The P-P-H part consists of over 2200 sentences from the Dutch daily newspaper Trouw from 2001. Average sentence length is about 16.5 tokens. The P-P-L part contains 1115 sentences taken from information brochures of Dutch Ministries. Average sentence length is about 18.5 tokens.

For training data, we used newspaper text from the TwNC (*Twente Newspaper*) corpus (Ordelman et al., 2007). We used Volkskrant 2001, NRC 2000, Algemeen Dagblad 1999. In addition, we used Volkskrant 1997 newspaper data extracted from the Volkskrant 1997 CDROM.

5.1 Results on Alpino Treebank

Figure 3 presents results obtained on the Alpino Treebank. In the graphs, the various filters are compared with the baseline variant of the parser. Each of the filters outperforms the default model for all given time-out values. In fact, the base-

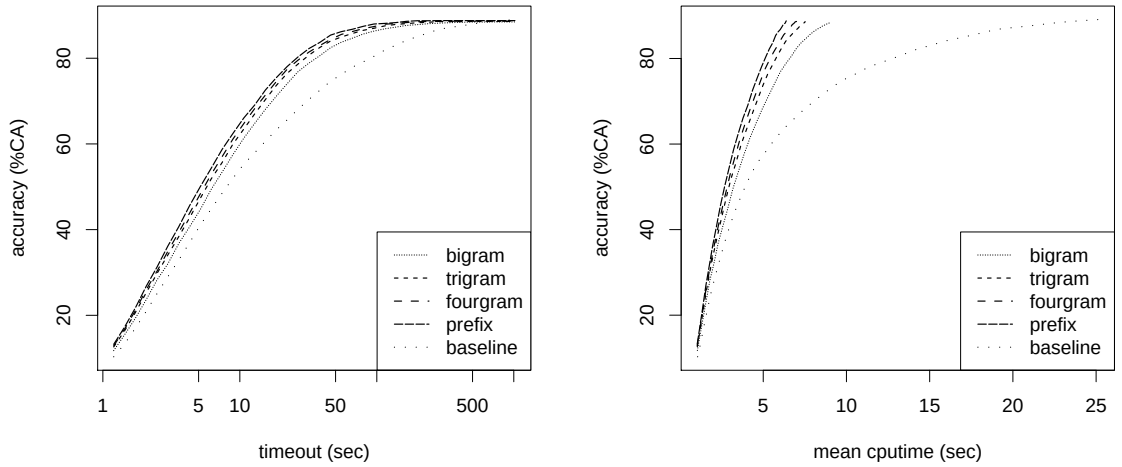


Figure 3: Accuracy versus time-out (on-line scenario), and accuracy versus mean CPU-time (off-line scenario) for various time-outs. The graphs compare the default setting of Alpino with the effect of the various filters based on all available training data. Evaluation on the Alpino treebank.

line parser improves upon the prefix filter only for unrealistic time-outs larger than fifteen minutes of CPU-time. The difference in accuracy for a given time-out value can be considerable: as much as 12% for time-outs around 30 seconds of CPU-time.

If we focus on mean CPU-time (off-line scenario), differences are even more pronounced. Without the filter, an accuracy of about 63% is obtained for a mean CPU-time of 6 seconds. The prefix filtering method obtains accuracy of more than 86% for the same mean CPU-time. For that level of accuracy, the base-line model requires a mean CPU-time of about 25 seconds. In other words, for the same level of accuracy, the prefix filter leads to a parser that is more than four times faster.

5.2 Effect of the amount of training data

In the first two graphs of figure 4 we observe the effect of the amount of training data. As can be expected, increasing the amount of data increases the accuracy, and decreases efficiency (because more derivation steps have been observed, hence fewer derivations are filtered out). Generally, models that take into account larger parts of the history require more data to obtain good accuracy, but they are also faster. For each of the variants, adding more training data after about 40 million words does not lead to much further improvement; the little improvement that is observed, is balanced by a slight increase in parse times too.

It is interesting to note that the accuracy of some of the filters improves slightly upon the baseline parser (without any filtering). This can be explained by the fact that the Alpino parser includes a best-first beam search to select the best parse from the parse forest. Apparently, in some cases the filter throws away candidate parses which would otherwise confuse this heuristic best search procedure.

5.3 Experiment with D-Coi data

In this section, we confirm the experimental results obtained on the Alpino Treebank by performing similar experiments on the D-Coi data. The purpose of this confirmation is twofold. On the one hand, the Alpino Treebank might not be a reliable test set for the Alpino parser, because it has been used quite intensively during the development of various components of the system. On the other hand, we might regard the experiments in the previous section as development experiments from which we learn the best parameters of the approach. The real evaluation of the technique is now performed using only the best method found on the development set, which is the *prefix* filter with $\tau = 0$.

We performed experiments with two parts of the D-Coi corpus. The first data set, P-P-H, contains newspaper data, and is therefore comparable both

with the Alpino Treebank, and more importantly, with the training data that we used to develop the filters. In order to check if the success of the filtering methods requires that training data and test data need to be taken from similar texts, we also provide experimental results on a test set consisting of different material: the P-P-L part of the D-Coi corpus, which contains text extracted from information brochures published by Dutch Ministries.

The third and fourth graphs in figure 4 provide results obtained on the P-P-H corpus. The increased efficiency of the prefix filter is slightly less pronounced. This may be due to the smaller mean sentence length of this data set. Still, the prefix filtering method performs much better for a large variety of time-outs. Only for very high, unrealistic, time-outs, the baseline parser obtains better accuracy. The same general trend is observed in the P-P-L data-set. From these results we tentatively conclude that the proposed technique is applicable across text types and domains.

6 Discussion

One may wonder how the technique introduced in this paper relates to techniques in which the disambiguation model is used directly during parsing to eliminate unlikely partial parses. An example in the context of wide coverage unification-based parsing is the beam thresholding technique employed in the Enju HPSG parser for English (Tsuruoka et al., 2004; Ninomiya et al., 2005).

In a beam-search parser, unlikely partial analyses are constructed, and then - based on the probability assigned to these partial analyses - removed from further consideration. One potential advantage of the use of our filters may be, that many of these partial analyses will not even be constructed in the first place, and therefore no time is spent on these alternatives at all.

We have not performed a detailed comparison, because the statistical model employed in Alpino contains some features which refer to arbitrary large parts of a parse. Such non-local features are not allowed in the Enju approach.

A parsing system may also combine both types of techniques. In that case there is room for further experimentation. For instance, during the learning phase, it may be beneficial to allow for a wider beam, to obtain more reliable filters. During testing, the beam can perhaps be smaller than usual, since the filters already rule out many of the competing parses.

The idea that corpora can be used to improve parsing efficiency was an important ingredient of a technique that was called *grammar specialization*. An overview of grammar specialization techniques is given in (Sima'an, 1999). For instance, Rayner and Carter (1996) use explanation-based learning to specialize a given general grammar to a specific domain. They report important efficiency gains (the parser is about three times faster), coupled with a mild reduction of coverage (5% loss).

In contrast to our approach in which no manual annotation is required, Rayner and Carter (1996) report that for each sentence in the training data, the best parse was selected manually from the set of parses generated by the parser. For the experiments described in the paper, this constituted an effort of two and a half person-months. As a consequence, they use only 15.000 training examples (taken from ATIS, so presumably relatively short sentences). In our experiments, we used up to 4 million sentences.

A further difference is related to the pruning strategies. Our pruning strategies are extremely simple. The cutting criteria employed in grammar specialization either require carefully manually tuning, or require more complicated statistical techniques (Samuelsson, 1994); automatically derived cutting criteria, however, perform considerably worse.

A possible improvement of our approach consists of predicting whether for a given input sentence the filter should be used, or whether the sentence appears to be 'easy' enough to allow for a full parse. For instance, one may chose to use the filter only for sentences of a given minimum length. Initial experiments indicate that such a setup may improve somewhat over the results presented here.

Acknowledgments

This research was carried out in part in the context of the STEVIN programme which is funded by the Dutch and Flemish governments (http://taalunieversum.org/taal/technologie/stevin/).

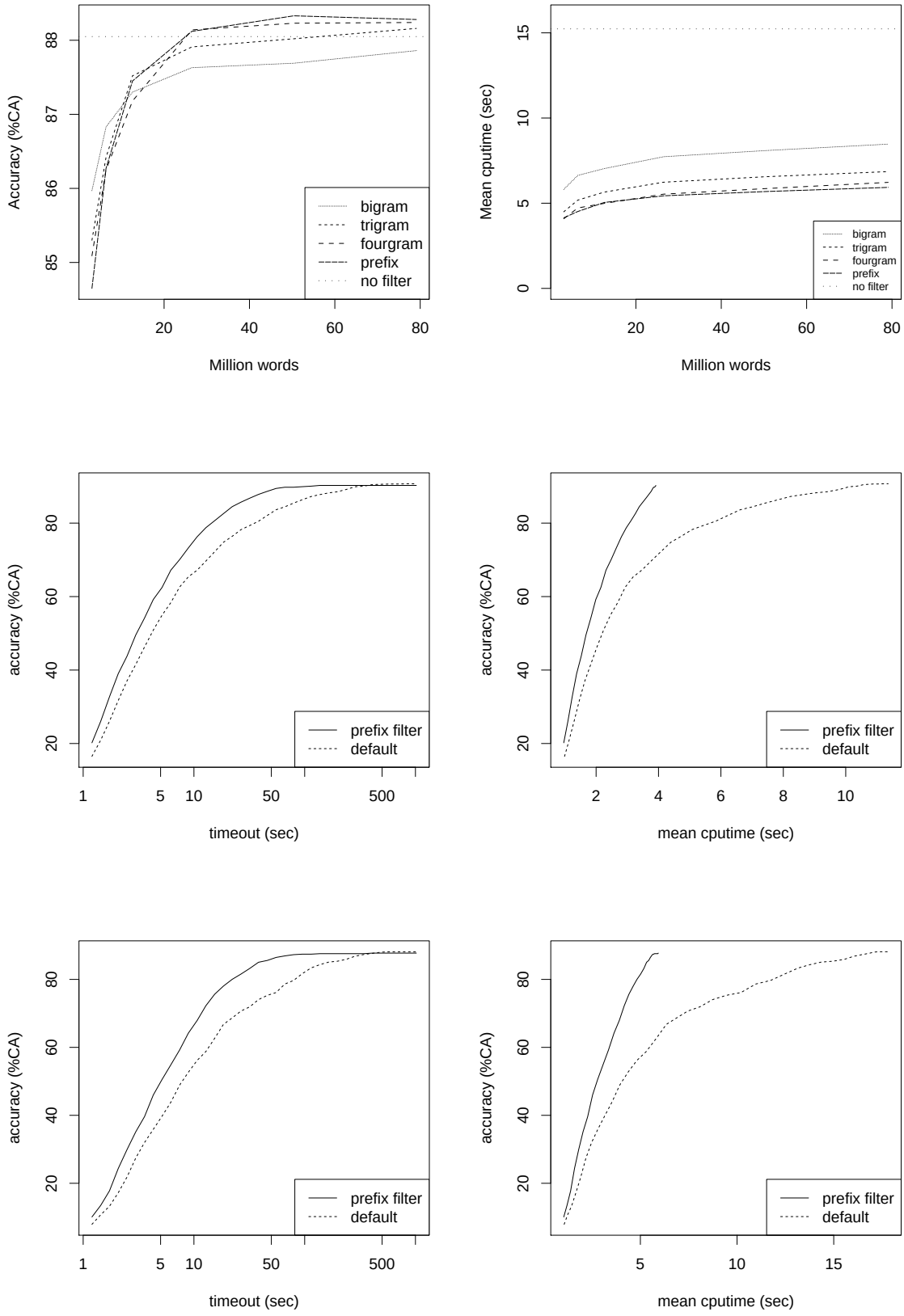


Figure 4: The first two graphs present accuracy (left) and mean CPU-time (right) as a function of the amount of training data used. Evaluation on 10% of the Alpino Treebank. The third and fourth graph present accuracy versus time-out, and accuracy versus mean CPU-time for various time-outs. The graph compares the baseline system with the parser which uses the prefix filter based on all available training data. Evaluation on the D-Coi P-P-H 1-109 data-set (newspaper text). The two last graphs are similar, based on the D-Coi P-P-L data-set (brochures).

References

P. C. Uit den Boogaart. 1975. *Woordfrequenties in geschreven en gesproken Nederlands*. Oosthoek, Scheltema & Holkema, Utrecht. Werkgroep Frequentie-onderzoek van het Nederlands.

Heleen Hoekstra, Michael Moortgat, Bram Renmans, Machteld Schouppe, Ineke Schuurman, and Ton van der Wouden, 2003. *CGN Syntactische Annotatie*, December.

Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. 1983. BUP: a bottom up parser embedded in Prolog. *New Generation Computing*, 1(2).

Takashi Ninomiya, Yoshimasa Tsuruoka, Yusuke Miyao, and Jun'ichi Tsujii. 2005. Efficacy of beam thresholding, unification filtering and hybrid parsing. In *Proceedings of the International Workshop on Parsing Technologies (IWPT)*.

Roeland Ordelman, Franciska de Jong, Arjan van Hessen, and Hendri Hondorp. 2007. Twnc: a multifaceted Dutch news corpus. *ELRA Newsletter*, 12(3/4):4–7.

Fernando C. N. Pereira and Stuart M. Shieber. 1987. *Prolog and Natural Language Analysis*. Center for the Study of Language and Information Stanford.

Manny Rayner and David Carter. 1996. Fast parsing using pruning and grammar specialization. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz.

Christer Samuelsson. 1994. Grammar specialization through entropy thresholds. In *32th Annual Meeting of the Association for Computational Linguistics*, New Mexico. ACL.

Khalil Sima'an. 1999. *Learning Efficient Disambiguation*. Ph.D. thesis, University of Utrecht.

Yoshimasa Tsuruoka, Yusuke Miyao, and Jun'ichi Tsujii. 2004. Towards efficient probabilistic hpsg parsing: integrating semantic and syntactic preference to guide the parsing. In *Beyond Shallow Analyses - Formalisms and statistical modeling for deep analyses*, Hainan China. IJCNLP.

Leonoor van der Beek, Gosse Bouma, Robert Malouf, and Gertjan van Noord. 2002. The Alpino dependency treebank. In *Computational Linguistics in the Netherlands*.

Gertjan van Noord, Ineke Schuurman, and Vincent Vandeghinste. 2006. Syntactic annotation of large corpora in STEVIN. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*, Genoa, Italy.

Gertjan van Noord. 1997. An efficient implementation of the head corner parser. *Computational Linguistics*, 23(3):425–456. cmp-lg/9701004.