A Classifier-Based Parser with Linear Run-Time Complexity

Kenji Sagae and **Alon Lavie**
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{sagae,alavie}@cs.cmu.edu

Abstract

We present a classifier-based parser that produces constituent trees in linear time. The parser uses a basic bottom-up shift-reduce algorithm, but employs a classifier to determine parser actions instead of a grammar. This can be seen as an extension of the deterministic dependency parser of Nivre and Scholz (2004) to full constituent parsing. We show that, with an appropriate feature set used in classification, a very simple one-path greedy parser can perform at the same level of accuracy as more complex parsers. We evaluate our parser on section 23 of the WSJ section of the Penn Treebank, and obtain precision and recall of 87.54% and 87.61%, respectively.

1 Introduction

Two classifier-based deterministic dependency parsers for English have been proposed recently (Nivre and Scholz, 2004; Yamada and Matsumoto, 2003). Although they use different parsing algorithms, and differ on whether or not dependencies are labeled, they share the idea of greedily pursuing a single path, following parsing decisions made by a classifier. Despite their greedy nature, these parsers achieve high accuracy in determining dependencies. Although state-of-the-art statistical parsers (Collins, 1997; Charniak, 2000) are more accurate, the simplicity and efficiency of deterministic parsers make them attractive in a number of situations requiring fast, light-weight parsing, or parsing of large amounts of data. However, dependency analyses lack important information contained in constituent structures. For example, the tree-path feature has been shown to be valuable in semantic role labeling (Gildea and Palmer, 2002).

We present a parser that shares much of the simplicity and efficiency of the deterministic dependency parsers, but produces both dependency and constituent structures simultaneously. Like the parser of Nivre and Scholz (2004), it uses the basic shift-reduce stack-based parsing algorithm, and runs in linear time. While it may seem that the larger search space of constituent trees (compared to the space of dependency trees) would make it unlikely that accurate parse trees could be built deterministically, we show that the precision and recall of constituents produced by our parser are close to those produced by statistical parsers with higher run-time complexity.

One desirable characteristic of our parser is its simplicity. Compared to other successful approaches to corpus-based constituent parsing, ours is remarkably simple to understand and implement. An additional feature of our approach is its modularity with regard to the algorithm and the classifier that determines the parser's actions. This makes it very simple for different classifiers and different sets of features to be used with the same parser with very minimal work. Finally, its linear run-time complexity allows our parser to be considerably faster than lexicalized PCFG-based parsers. On the other hand, a major drawback of the classifier-based parsing framework is that, depending on

Proceedings of the Ninth International Workshop on Parsing Technologies (IWPT), pages 125–132,
Vancouver, October 2005.

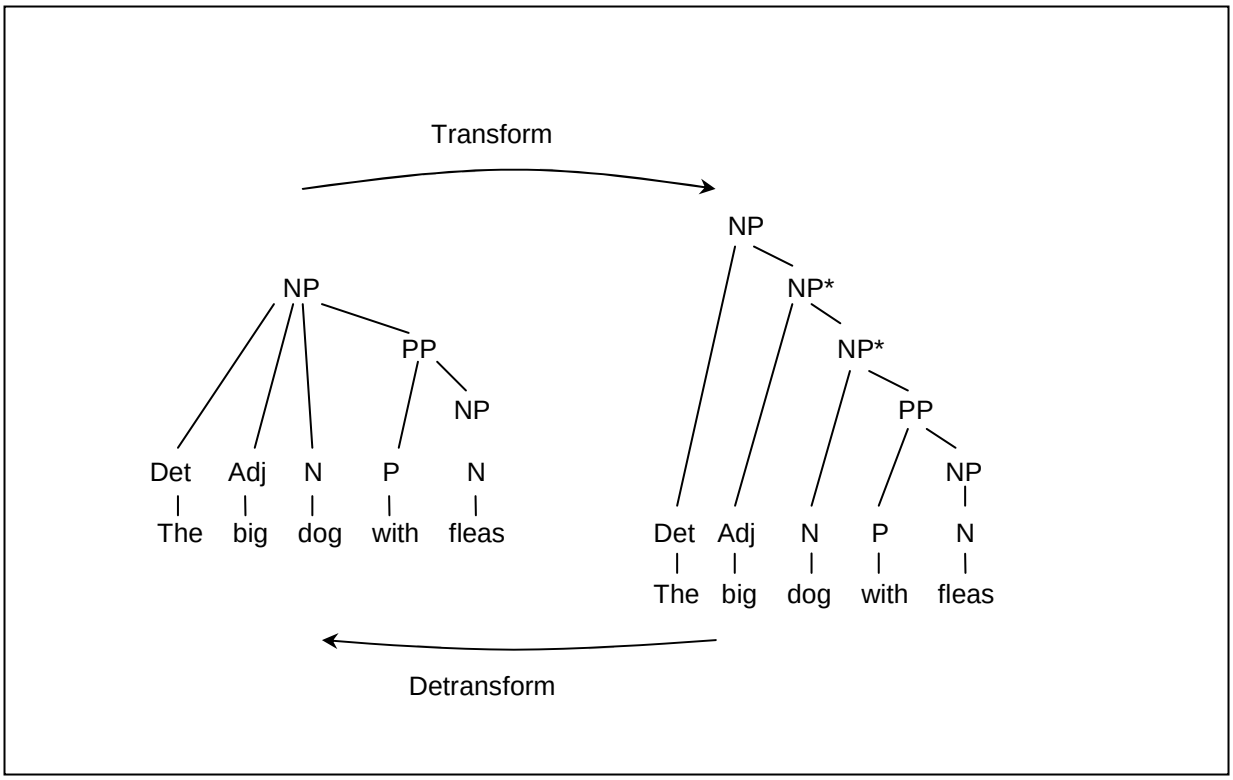

Figure 1: An example of the binarization transform/detransform. The original tree (left) has one node (NP) with four children. In the transformed tree, internal structure (marked by nodes with asterisks) was added to the subtree rooted by the node with more than two children. The word "dog" is the head of the original NP, and it is kept as the head of the transformed NP, as well as the head of each NP* node.

the classifier used, its training time can be much longer than that of other approaches.

Like other deterministic parsers (and unlike many statistical parsers), our parser considers the problem of syntactic analysis separately from part-of-speech (POS) tagging. Because the parser greedily builds trees bottom-up in one pass, considering only one path at any point in the analysis, the task of assigning POS tags to words is done before other syntactic analysis. In this work we focus only on the processing that occurs once POS tagging is completed. In the sections that follow, we assume that the input to the parser is a sentence with corresponding POS tags for each word.

2 Parser Description

Our parser employs a basic bottom-up shift-reduce parsing algorithm, requiring only a single pass over the input string. The algorithm considers only trees with unary and binary branching. In order to use trees with arbitrary branching for training, or generating them with the parser, we employ an instance of the transformation/detransformation process described in (Johnson, 1998). In our case, the transformation step involves simply converting each production with n children (where $n > 2$) into $n - 1$ binary productions. Trees must be lexicalized[1], so that the newly created internal structure of constituents with previous branching of more than two contains only subtrees with the same lexical head as the original constituent. Additional non-terminal symbols introduced in this process are clearly marked. The transformed (or "binarized") trees may then be used for training. Detransformation is applied to trees produced by the parser. This involves the removal of non-terminals intro-

[1] If needed, constituent head-finding rules such as those mentioned in Collins (1996) may be used.

duced in the transformation process, producing trees with arbitrary branching. An example of transformation/detransformation is shown in figure 1.

2.1 Algorithm Outline

The parsing algorithm involves two main data structures: a stack *S*, and a queue *W*. Items in *S* may be terminal nodes (POS-tagged words), or (lexicalized) subtrees of the final parse tree for the input string. Items in *W* are terminals (words tagged with parts-of-speech) corresponding to the input string. When parsing begins, *S* is empty and *W* is initialized by inserting every word from the input string in order, so that the first word is in front of the queue.

Only two general actions are allowed: shift and reduce. A shift action consists only of removing (shifting) the first item (POS-tagged word) from *W* (at which point the next word becomes the new first item), and placing it on top of *S*. Reduce actions are subdivided into unary and binary cases. In a unary reduction, the item on top of *S* is popped, and a new item is pushed onto *S*. The new item consists of a tree formed by a non-terminal node with the popped item as its single child. The lexical head of the new item is the same as the lexical head of the popped item. In a binary reduction, two items are popped from *S* in sequence, and a new item is pushed onto *S*. The new item consists of a tree formed by a non-terminal node with two children: the first item popped from *S* is the right child, and the second item is the left child. The lexical head of the new item is either the lexical head of its left child, or the lexical head of its right child.

If *S* is empty, only a shift action is allowed. If *W* is empty, only a reduce action is allowed. If both *S* and *W* are non-empty, either shift or reduce actions are possible. Parsing terminates when *W* is empty and *S* contains only one item, and the single item in *S* is the parse tree for the input string. Because the parse tree is lexicalized, we also have a dependency structure for the sentence. In fact, the binary reduce actions are very similar to the reduce actions in the dependency parser of Nivre and Scholz (2004), but they are executed in a different order, so constituents can be built. If *W* is empty, and more than one item remain in *S*, and no further reduce actions take place, the input string is rejected.

2.2 Determining Actions with a Classifier

A parser based on the algorithm described in the previous section faces two types of decisions to be made throughout the parsing process. The first type concerns whether to shift or reduce when both actions are possible, or whether to reduce or reject the input when only reduce actions are possible. The second type concerns what syntactic structures are created. Specifically, what new non-terminal is introduced in unary or binary reduce actions, or which of the left or right children are chosen as the source of the lexical head of the new subtree produced by binary reduce actions. Traditionally, these decisions are made with the use of a grammar, and the grammar may allow more than one valid action at any single point in the parsing process. When multiple choices are available, a grammar-driven parser may make a decision based on heuristics or statistical models, or pursue every possible action following a search strategy. In our case, both types of decisions are made by a classifier that chooses a unique action at every point, based on the local context of the parsing action, with no explicit grammar. This type of classifier-based parsing where only one path is pursued with no backtracking can be viewed as greedy or deterministic.

In order to determine what actions the parser should take given a particular parser configuration, a classifier is given a set of features derived from that configuration. This includes, crucially, the two topmost items in the stack *S*, and the item in front of the queue *W*. Additionally, a set of context features is derived from a (fixed) limited number of items below the two topmost items of *S*, and following the item in front of *W*. The specific features are shown in figure 2.

The classifier's target classes are parser actions that specify both types of decisions mentioned above. These classes are:

- SHIFT: a shift action is taken;
- REDUCE-UNARY-*XX*: a unary reduce action is taken, and the root of the new subtree pushed onto *S* is of type *XX* (where *XX* is a non-terminal symbol, typically NP, VP, PP, for example);
- REDUCE-LEFT-*XX*: a binary reduce action is taken, and the root of the new subtree pushed onto *S* is of non-terminal type *XX*.

Let:

S(n) denote the nth item from the top of the stack *S*, and
W(n) denote the nth item from the front of the queue *W*.

Features:

- The head-word (and its POS tag) of: S(0), S(1), S(2), and S(3)
- The head-word (and its POS tag) of: W(0), W(1), W(3) and W(3)
- The non-terminal node of the root of: S(0), and S(1)
- The non-terminal node of the left child of the root of: S(0), and S(1)
- The non-terminal node of the right child of the root of: S(0), and S(1)
- The non-terminal node of the left child of the root of: S(0), and S(1)
- The non-terminal node of the left child of the root of: S(0), and S(1)
- The linear distance (number of words apart) between the head-words of S(0) and S(1)
- The number of lexical items (words) that have been found (so far) to be dependents of the head-words of: S(0), and S(1)
- The most recently found lexical dependent of the head of the head-word of S(0) that is to the left of S(0)'s head
- The most recently found lexical dependent of the head of the head-word of S(0) that is to the right of S(0)'s head
- The most recently found lexical dependent of the head of the head-word of S(0) that is to the left of S(1)'s head
- The most recently found lexical dependent of the head of the head-word of S(0) that is to the right of S(1)'s head

Figure 2: Features used for classification. The features described in items 1 – 7 are more directly related to the lexicalized constituent trees that are built during parsing, while the features described in items 8 – 13 are more directly related to the dependency structures that are built simultaneously to the constituent structures.

Additionally, the head of the new subtree is the same as the head of the left child of the root node;

- REDUCE-RIGHT-*XX*: a binary reduce action is taken, and the root of the new subtree pushed onto *S* is of non-terminal type *XX*. Additionally, the head of the new subtree is the same as the head of the right child of the root node.

2.3 A Complete Classifier-Based Parser than Runs in Linear Time

When the algorithm described in section 2.1 is combined with a trained classifier that determines its parsing actions as described in section 2.2, we have a complete classifier-based parser. Training the parser is accomplished by training its classifier. To that end, we need training instances that consist of sets of features paired with their classes corre-

sponding to the correct parsing actions. These instances can be obtained by running the algorithm on a corpus of sentences for which the correct parse trees are known. Instead of using the classifier to determine the parser's actions, we simply determine the correct action by consulting the correct parse trees. We then record the features and corresponding actions for parsing all sentences in the corpus into their correct trees. This set of features and corresponding actions is then used to train a classifier, resulting in a complete parser.

When parsing a sentence with n words, the parser takes n shift actions (exactly one for each word in the sentence). Because the maximum branching factor of trees built by the parser is two, the total number of binary reduce actions is $n - 1$, if a complete parse is found. If the input string is rejected, the number of binary reduce actions is less than $n - 1$. Therefore, the number of shift and binary reduce actions is linear with the number of words in the input string. However, the parser as described so far has no limit on the number of unary reduce actions it may take. Although in practice a parser properly trained on trees reflecting natural language syntax would rarely make more than $2n$ unary reductions, pathological cases exist where an infinite number of unary reductions would be taken, and the algorithm would not terminate. Such cases may include the observation in the training data of sequences of unary productions that cycle through (repeated) non-terminals, such as *A->B->A->B*. During parsing, it is possible that such a cycle may be repeated infinitely.

This problem can be easily prevented by limiting the number of consecutive unary reductions that may be made to a finite number. This may be the number of non-terminal types seen in the training data, or the length of the longest chain of unary productions seen in the training data. In our experiments (described in section 3), we limited the number of consecutive unary reductions to three, although the parser never took more than two unary reduction actions consecutively in any sentence. When we limit the number of consecutive unary reductions to a finite number m, the parser makes at most $(2n - 1)m$ unary reductions when parsing a sentence of length n. Placing this limit not only guarantees that the algorithm terminates, but also guarantees that the number of actions taken by the parser is $O(n)$, where n is the length of the input string. Thus, the parser runs in linear time, assuming that classifying a parser action is done in constant time.

3 Similarities to Previous Work

As mentioned before, our parser shares similarities with the dependency parsers of Yamada and Matsumoto (2003) and Nivre and Scholz (2004) in that it uses a classifier to guide the parsing process in deterministic fashion. While Yamada and Matsumoto use a quadratic run-time algorithm with multiple passes over the input string, Nivre and Scholz use a simplified version of the algorithm described here, which handles only (labeled or unlabeled) dependency structures.

Additionally, our parser is in some ways similar to the maximum-entropy parser of Ratnaparkhi (1997). Ratnaparkhi's parser uses maximum-entropy models to determine the actions of a shift-reduce-like parser, but it is capable of pursuing several paths and returning the top-K highest scoring parses for a sentence. Its observed time is linear, but parsing is somewhat slow, with sentences of length 20 or more taking more than one second to parse, and sentences of length 40 or more taking more than three seconds. Our parser only pursues one path per sentence, but it is very fast and of comparable accuracy (see section 4). In addition, Ratnaparkhi's parser uses a more involved algorithm that allows it to work with arbitrary branching trees without the need of the binarization transform employed here. It breaks the usual reduce actions into smaller pieces (CHECK and BUILD), and uses two separate passes (not including the POS tagging pass) for determining chunks and higher syntactic structures separately.

Finally, there have been other deterministic shift-reduce parsers introduced recently, but their levels of accuracy have been well below the state-of-the-art. The parser in Kalt (2004) uses a similar algorithm to the one described here, but the classification task is framed differently. Using decision trees and fewer features, Kalt's parser has significantly faster training and parsing times, but its accuracy is much lower than that of our parser. Kalt's parser achieves precision and recall of about 77% and 76%, respectively (with automatically tagged text), compared to our parser's 86% (see section 4). The parser of Wong and Wu (1999) uses a separate NP-chunking step and, like Ratnaparkhi's parser, does not require a binary trans-

	Precision	Recall	Dependency	Time (min)
Charniak	89.5	89.6	92.1	28
Collins	88.3	88.1	91.5	45
Ratnaparkhi	87.5	86.3	Unk	Unk
Y&M	-	-	90.3	Unk
N&S	-	-	87.3	21
MBLpar	80.0	80.2	86.3	127
SVMpar	87.5	87.6	90.3	11

Table 1: Summary of results on labeled precision and recall of constituents, dependency accuracy, and time required to parse the test set. The parsers of Yamada and Matsumoto (Y&M) and Nivre and Scholz (N&S) do not produce constituent structures, only dependencies. "unk" indicates unknown values. Results for MBLpar and SVMpar using correct POS tags (if automatically produced POS tags are used, accuracy figures drop about 1.5% over all metrics).

form. It achieves about 81% precision and 82% recall with gold-standard tags (78% and 79% with automatically tagged text). Wong and Wu's parser is further differentiated from the other parsers mentioned here in that it does not use lexical items, working only from part-of-speech tags.

4 Experiments

We conducted experiments with the parser described in section 2 using two different classifiers: TinySVM (a support vector machine implementation by Taku Kudo)[2], and the memory-based learner TiMBL (Daelemans et al., 2004). We trained and tested the parser on the Wall Street Journal corpus of the Penn Treebank (Marcus et al., 1993) using the standard split: sections 2-21 were used for training, section 22 was used for development and tuning of parameters and features, and section 23 was used for testing. Every experiment reported here was performed on a Pentium IV 1.8GHz with 1GB of RAM.

Each tree in the training set had empty-node and function tag information removed, and the trees were lexicalized using similar head-table rules as those mentioned in (Collins, 1996). The trees were then converted into trees containing only unary and binary branching, using the binarization transform described in section 2. Classifier training instances of features paired with classes (parser actions) were extracted from the trees in the training set, as described in section 2.3. The total number of training instances was about 1.5 million.

The classifier in the SVM-based parser (denoted by SVMpar) uses the polynomial kernel with degree 2, following the work of Yamada and Matsumoto (2003) on SVM-based deterministic dependency parsing, and a one-against-all scheme for multi-class classification. Because of the large number of training instances, we used Yamada and Matsumoto's idea of splitting the training instances into several parts according to POS tags, and training classifiers on each part. This greatly reduced the time required to train the SVMs, but even with the splitting of the training set, total training time was about 62 hours. Training set splitting comes with the cost of reduction in accuracy of the parser, but training a single SVM would likely take more than one week. Yamada and Matsumoto experienced a reduction of slightly more than 1% in de-

[2] http://chasen.org/~taku/software/TinySVM

pendency accuracy due to training set splitting, and we expect that a similar loss is incurred here.

When given perfectly tagged text (gold tags extracted from the Penn Treebank), SVMpar has labeled constituent precision and recall of 87.54% and 87.61%, respectively, and dependency accuracy of 90.3% over all sentences in the test set. The total time required to parse the entire test set was 11 minutes. Out of more than 2,400 sentences, only 26 were rejected by the parser (about 1.1%). For these sentences, partial analyses were created by combining the items in the stack in flat structures, and these were included in the evaluation. Predictably, the labeled constituent precision and recall obtained with automatically POS-tagged sentences were lower, at 86.01% and 86.15%. The part-of-speech tagger used in our experiments was SVMTool (Giménez and Márquez, 2004), and its accuracy on the test set is 97%.

The MBL-based parser (denoted by MBLpar) uses the IB1 algorithm, with five nearest neighbors, and the modified value difference metric (MVDM), following the work of Nivre and Scholz (2004) on MBL-based deterministic dependency parsing. MBLpar was trained with all training instances in under 15 minutes, but its accuracy on the test set was much lower than that of SVMpar, with constituent precision and recall of 80.0% and 80.2%, and dependency accuracy of 86.3% (24 sentences were rejected). It was also much slower than SVMpar in parsing the test set, taking 127 minutes. In addition, the total memory required for running MBLpar (including the classifier) was close to 1 gigabyte (including the trained classifier), while SVMpar required only about 200 megabytes (including all the classifiers).

Table 1 shows a summary of the results of our experiments with SVMpar and MBLpar, and also results obtained with the Charniak (2000) parser, the Bikel (2003) implementation of the Collins (1997) parser, and the Ratnaparkhi (1997) parser. We also include the dependency accuracy from Yamada and Matsumoto's (2003) SVM-based dependency parser, and Nivre and Scholz's (2004) MBL-based dependency parser. These results show that the choice of classifier is extremely important in this task. SVMpar and MBLpar use the same algorithm and features, and differ only on the classifiers used to make parsing decisions. While in many natural language processing tasks different classifiers perform at similar levels of accuracy, we have observed a dramatic difference between using support vector machines and a memory-based learner. Although the reasons for such a large disparity in results is currently the subject of further investigation, we speculate that a relatively small difference in initial classifier accuracy results in larger differences in parser performance, due to the deterministic nature of the parser (certain errors may lead to further errors). We also believe classifier choice to be one major source of the difference in accuracy between Nivre and Scholz's parser and Yamada and Matsumoto's parser.

While the accuracy of SVMpar is below that of lexicalized PCFG-based statistical parsers, it is surprisingly good for a greedy parser that runs in linear time. Additionally, it is considerably faster than lexicalized PCFG-based parsers, and offers a good alternative for when fast parsing is needed. MBLpar, on the other hand, performed poorly in terms of accuracy and speed.

5 Conclusion and Future Work

We have presented a simple shift-reduce parser that uses a classifier to determine its parsing actions and runs in linear time. Using SVMs for classification, the parser has labeled constituent precision and recall higher than 87% when using the correct part-of-speech tags, and slightly higher than 86% when using automatically assigned part-of-speech tags. Although its accuracy is not as high as those of state-of-the-art statistical parsers, our classifier-based parser is considerably faster than several well-known parsers that employ search or dynamic programming approaches. At the same time, it is significantly more accurate than previously proposed deterministic parsers for constituent structures.

We have also shown that much of the success of a classifier-based parser depends on what classifier is used. While this may seem obvious, the differences observed here are much greater than what would be expected from looking, for example, at results from chunking/shallow parsing (Zhang et al., 2001; Kudo and Matsumoto, 2001; Veenstra and van den Bosch, 2000).

Future work includes the investigation of the effects of individual features, the use of additional classification features, and the use of different classifiers. In particular, the use of tree features seems appealing. This may be accomplished with SVMs

using a tree kernel, or the tree boosting classifier BACT described in (Kudo and Matsumoto, 2004). Additionally, we plan to investigate the use of the beam strategy of Ratnaparkhi (1997) to pursue multiple parses while keeping the run-time linear.

References

Charniak, E. 2000. A maximum-entropy-inspired parser. *Proceedings of the First Annual Meeting of the North American Chapter of the Association for Computational Linguistics*. Seattle, WA.

Collins, M. 1997. Three generative, lexicalized models for statistical parsing. *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics* (pp. 16-23). Madrid, Spain.

Daelemans, W., Zavrel, J., van der Sloot, K., and van den Bosch, A. 2004. TiMBL: Tilburg Memory Based Learner, version 5.1, reference guide. *ILK Research Group Technical Report Series* no. 04-02, 2004.

Gildea, D., and Palmer, M. 2002. The necessity of syntactic parsing for predicate argument recognition. *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (pp. 239-246). Philadelphia, PA.

Kalt, T. 2004. Induction of greedy controllers for deterministic treebank parsers. *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*. Barcelona, Spain.

Kudo, T., and Matsumoto, Y. 2004. A boosting algorithm for classification of semi-structured text. *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*. Barcelona, Spain.

Kudo, T., and Matsumoto, Y. 2001. Chunking with support vector machines. *Proceedings of the Second Meeting of the North American Chapter of the Association for Computational Linguistics*. Pittsburgh, PA.

Johnson, M. 1998. PCFG models of linguistic tree representations. *Computational Linguistics*, 24:613-632.

Marcus, M. P., Santorini, B., and Marcinkiewics, M. A. 1993. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19.

Nivre, J., and Scholz, M. 2004. Deterministic dependency parsing of English text. *Proceedings of the 20th International Conference on Computational Linguistics* (pp. 64-70). Geneva, Switzerland.

Ratnaparkhi, A. 1997. A linear observed time statistical parser based on maximum entropy models. *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*. Providence, Rhode Island.

Veenstra, J., van den Bosch, A. 2000. Single-classifier memory-based phrase chunking. *Proceedings of Fourth Workshop on Computational Natural Language Learning (CoNLL 2000).* Lisbon, Portugal.

Wong, A., and Wu. D. 1999. Learning a lightweight robust deterministic parser. *Proceedings of the Sixth European Conference on Speech Communication and Technology*. Budapest.

Yamada, H., and Matsumoto, Y. 2003. Statistical dependency analysis with support vector machines. *Proceedings of the Eighth International Workshop on Parsing Technologies.* Nancy, France.

Zhang, T., Damerau, F., and Johnson, D. 2002. Text chunking using regularized winnow. *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics.* Tolouse, France.