cmp-lg/9706014 11 Jun 1997

A Linear Observed Time Statistical Parser Based on Maximum Entropy Models

Adwait Ratnaparkhi*
Dept. of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389
adwait@unagi.cis.upenn.edu

Abstract

This paper presents a statistical parser for natural language that obtains a parsing accuracy—roughly 87% precision and 86% recall—which surpasses the best previously published results on the Wall St. Journal domain. The parser itself requires very little human intervention, since the information it uses to make parsing decisions is specified in a concise and simple manner, and is combined in a fully automatic way under the maximum entropy framework. The observed running time of the parser on a test sentence is linear with respect to the sentence length. Furthermore, the parser returns several scored parses for a sentence, and this paper shows that a scheme to pick the best parse from the 20 highest scoring parses *could* yield a dramatically higher accuracy of 93% precision and recall.

1 Introduction

This paper presents a statistical parser for natural language that finds one or more scored syntactic parse trees for a given input sentence. The parsing accuracy—roughly 87% precision and 86% recall—surpasses the best previously published results on the Wall St. Journal domain. The parser consists of the following three conceptually distinct parts:

1. A set of procedures that use certain actions to incrementally construct parse trees.
2. A set of maximum entropy models that compute probabilities of the above actions, and effectively "score" parse trees.
3. A search heuristic which attempts to find the highest scoring parse tree for a given input sentence.

The maximum entropy models used here are similar in form to those in (Ratnaparkhi, 1996; Berger, Della Pietra, and Della Pietra, 1996; Lau, Rosenfeld, and Roukos, 1993). The models compute the probabilities of actions based on certain syntactic characteristics, or *features*, of the current context. The features used here are defined in a concise and simple manner, and their relative importance is determined automatically by applying a training procedure on a corpus of syntactically annotated sentences, such as the Penn Treebank (Marcus, Santorini, and Marcinkiewicz, 1994). Although creating the annotated corpus requires much linguistic expertise, creating the feature set for the parser itself requires very little linguistic effort.

Also, the search heuristic is very simple, and its observed running time on a test sentence is linear with respect to the sentence length. Furthermore, the search heuristic returns several scored parses for a sentence, and this paper shows that a scheme to pick the best parse from the 20 highest scoring parses *could* yield a dramatically higher accuracy of 93% precision and recall.

Sections 2, 3, and 4 describe the tree-building procedures, the maximum entropy models, and the search heuristic, respectively. Section 5 describes experiments with the Penn Treebank and section 6 compares this paper with previously published works.

2 Procedures for Building Trees

The parser uses four procedures, TAG, CHUNK, BUILD, and CHECK, that incrementally build parse trees with their actions. The procedures are applied in three left-to-right passes over the input sentence; the first pass applies TAG, the second pass applies CHUNK, and the third pass applies BUILD and CHECK. The passes, the procedures they apply, and the actions of the procedures are summarized in table 1 and described below.

The actions of the procedures are designed so that any possible complete parse tree T for the input sentence corresponds to *exactly one* sequence of actions; call this sequence the *derivation* of T. Each procedure, when given a derivation $d = \{a_1 \ldots a_n\}$, pre-

* The author acknowledges the support of ARPA grant N66001-94C-6043.

Pass	Procedure	Actions	Description
First Pass	TAG	A POS tag in tag set	Assign POS Tag to word
Second Pass	CHUNK	`Start X`, `Join X`, `Other`	Assign Chunk tag to POS tag and word
Third Pass	BUILD	`Start X`, `Join X`, where `X` is a constituent label in label set	Assign current tree to start a new constituent, or to join the previous one
	CHECK	`Yes`, `No`	Decide if current constituent is complete

Table 1: Tree-Building Procedures of Parser

dicts some action a_{n+1} to create a new derivation $d' = \{a_1 \ldots a_{n+1}\}$. Typically, the procedures postulate many different values for a_{n+1}, which cause the parser to explore many different derivations when parsing an input sentence. But for demonstration purposes, figures 1–7 trace one possible derivation for the sentence "I saw the man with the telescope", using the part-of-speech (POS) tag set and constituent label set of the Penn treebank.

2.1 First Pass

The first pass takes an input sentence, shown in figure 1, and uses TAG to assign each word a POS tag. The result of applying TAG to each word is shown in figure 2.

2.2 Second Pass

The second pass takes the output of the first pass and uses CHUNK to determine the "flat" phrase chunks of the sentence, where a phrase is "flat" if and only if it is a constituent whose children consist solely of POS tags. Starting from the left, CHUNK assigns each (word,POS tag) pair a "chunk" tag, either `Start X`, `Join X`, or `Other`. Figure 3 shows the result after the second pass. The chunk tags are then used for chunk detection, in which any consecutive sequence of words $w_m \ldots w_n$ $(m \leq n)$ are grouped into a "flat" chunk X if w_m has been assigned `Start X` and $w_{m+1} \ldots w_n$ have all been assigned `Join X`. The result of chunk detection, shown in figure 4, is a forest of trees and serves as the input to the third pass.

2.3 Third Pass

The third pass always alternates between the use of BUILD and CHECK, and completes any remaining constituent structure. BUILD decides whether a tree will start a new constituent or join the incomplete constituent immediately to its left. Accordingly, it annotates the tree with either `Start X`, where `X` is any constituent label, or with `Join X`, where `X` matches the label of the incomplete constituent to the left. BUILD always processes the leftmost tree without any `Start X` or `Join X` annotation. Figure 5 shows an application of BUILD in which the action is `Join VP`. After BUILD, control passes to CHECK, which finds the most recently proposed constituent, and decides if it is complete. The most recently proposed constituent, shown in figure 6, is the rightmost sequence of trees $t_m \ldots t_n$ $(m \leq n)$ such that t_m is annotated with `Start X` and $t_{m+1} \ldots t_n$ are annotated with `Join X`. If CHECK decides `yes`, then the proposed constituent takes its place in the forest as an actual constituent, on which BUILD does its work. Otherwise, the constituent is not finished and BUILD processes the next tree in the forest, t_{n+1}. CHECK always answers `no` if the proposed constituent is a "flat" chunk, since such constituents must be formed in the second pass. Figure 7 shows the result when CHECK looks at the proposed constituent in figure 6 and decides `No`. The third pass terminates when CHECK is presented a constituent that spans the entire sentence.

Procedure	Actions	Similar Shift-Reduce Parser Action
CHECK	`No`	shift
CHECK	`Yes`	reduce α, where α is CFG rule of proposed constituent
BUILD	`Start X`, `Join X`	Determines α for subsequent reduce operations

Table 2: Comparison of BUILD and CHECK to operations of a shift-reduce parser

Table 2 compares the actions of BUILD and CHECK to the operations of a standard shift-reduce parser. The `No` and `Yes` actions of CHECK correspond to the shift and reduce actions, respectively. The important difference is that while a shift-reduce parser creates a constituent in one step (reduce α), the procedures BUILD and CHECK create it over several steps in smaller increments.

I saw the man with the telescope

Figure 1: Initial Sentence

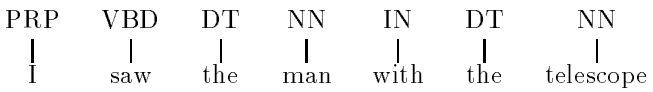

Figure 2: The result after First Pass

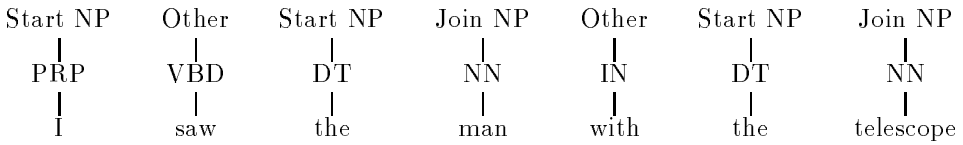

Figure 3: The result after Second Pass

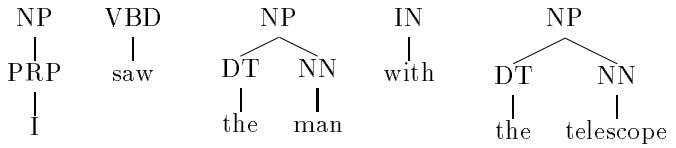

Figure 4: The result of chunk detection

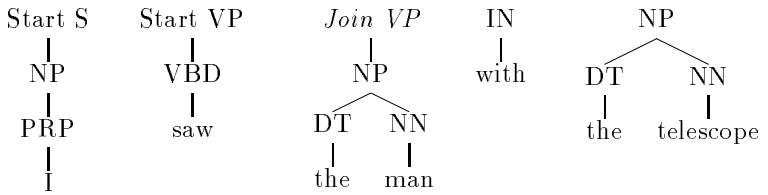

Figure 5: An application of BUILD in which `Join VP` is the action

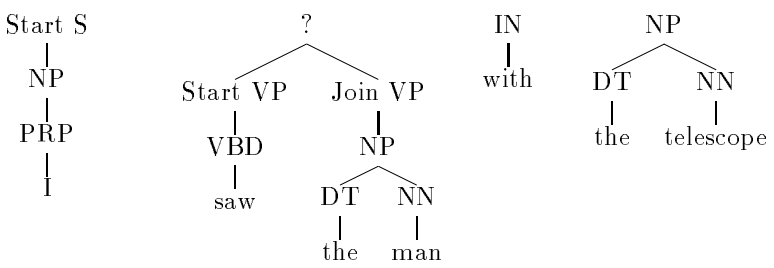

Figure 6: The most recently proposed constituent (shown under ?)

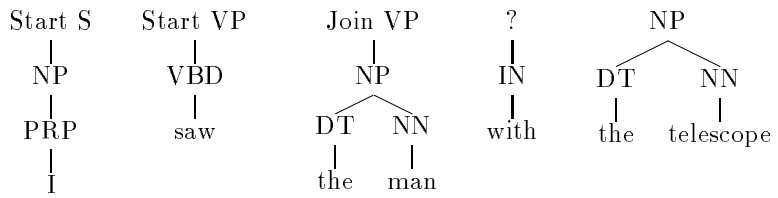

Figure 7: An application of CHECK in which `No` is the action, indicating that the proposed constituent in figure 6 is *not* complete. BUILD will now process the tree marked with ?

3 Probability Model

This paper takes a "history-based" approach (Black et al., 1993) where each tree-building procedure uses a probability model $p(a|b)$, derived from $p(a,b)$, to weight any action a based on the available context, or history, b. First, we present a few simple categories of contextual predicates that capture any information in b that is useful for predicting a. Next, the predicates are used to extract a set of features from a corpus of manually parsed sentences. Finally, those features are combined under the maximum entropy framework, yielding $p(a,b)$.

3.1 Contextual Predicates

Contextual predicates are functions that check for the presence or absence of useful information in a context b and return true or false accordingly. The comprehensive guidelines, or templates, for the contextual predicates of each tree building procedure are given in table 3. The templates use indices relative to the tree that is currently being modified. For example, if the current tree is the 5th tree, $\mathsf{cons}(-2)$ looks at the constituent label, head word, and start/join annotation of the 3rd tree in the forest. The actual contextual predicates are generated automatically by scanning the derivations of the trees in the manually parsed corpus with the templates. For example, an actual contextual predicate based on the template $\mathsf{cons}(0)$ might be "Does $\mathsf{cons}(0)$ = { `NP, he` } ?" Constituent head words are found, when necessary, with the algorithm in (Magerman, 1995).

Contextual predicates which look at head words, or especially pairs of head words, may not be reliable predictors for the procedure actions due to their sparseness in the training sample. Therefore, for each lexically based contextual predicate, there also exist one or more corresponding less specific, or "backed-off", contextual predicates which look at the same context, but *omit* one or more words. For example, the contexts $\mathsf{cons}(0,1^*)$, $\mathsf{cons}(0^*,1)$, $\mathsf{cons}(0^*,1^*)$ are the same as $\mathsf{cons}(0,1)$ but omit references to the head word of the 1st tree, the 0th tree, and both the 0th and 1st tree, respectively. The backed-off contextual predicates should allow the model to provide reliable probability estimates when the words in the history are rare. Backed-off predicates are not enumerated in table 3, but their existence is indicated with a * and †.

3.2 Maximum Entropy Framework

The contextual predicates derived from the templates of table 3 are used to create the features necessary for the maximum entropy models. The predicates for TAG, CHUNK, BUILD, and CHECK are used to scan the derivations of the trees in the corpus to form the training samples $\mathcal{T}_{\text{tag}}$, $\mathcal{T}_{\text{chunk}}$, $\mathcal{T}_{\text{build}}$, and $\mathcal{T}_{\text{check}}$, respectively. Each training sample has the form $\mathcal{T} = \{(a_1,b_1),(a_2,b_2),\ldots,(a_N,b_N)\}$, where a_i is an action of the corresponding procedure and b_i is the list of contextual predicates that were `true` in the context in which a_i was decided.

The training samples are respectively used to create the models p_{tag}, p_{chunk}, p_{build}, and p_{check}, all of which have the form:

$$p(a,b) = \pi \prod_{j=1}^{k} \alpha_j^{f_j(a,b)} \qquad (1)$$

where a is some action, b is some context, π is a normalization constant, α_j are the model parameters, $0 < \alpha_j < \infty$, and $f_j(a,b) \in \{0,1\}$ are called *features*, $j = \{1 \ldots k\}$. Features encode an action a' as well as some contextual predicate cp that a tree-building procedure would find useful for predicting the action a'. Any contextual predicate cp derived from table 3 which occurs 5 or more times in a training sample with a particular action a' is used to construct a feature f_j:

$$f_j(a,b) = \begin{cases} 1 & \text{if } cp(b) = \texttt{true} \;\&\&\; a = a' \\ 0 & \text{otherwise} \end{cases}$$

for use in the corresponding model. Each feature f_j corresponds to a parameter α_j, which can be viewed as a "weight" that reflects the importance of the feature.

The parameters $\{\alpha_1 \ldots \alpha_n\}$ are found automatically with *Generalized Iterative Scaling* (Darroch and Ratcliff, 1972), or GIS. The GIS procedure, as well as the maximum entropy and maximum likelihood properties of the distribution of form (1), are described in detail in (Ratnaparkhi, 1997). In general, the maximum entropy framework puts no limitations on the kinds of features in the model; no

Model	Categories	Description	Templates Used
TAG	See (Ratnaparkhi, 1996)		
CHUNK	chunkandpostag$(n)^*$	The word, POS tag, and chunk tag of nth leaf. Chunk tag omitted if $n \geq 0$.	chunkandpostag(0), chunkandpostag(-1), chunkandpostag(-2) chunkandpostag(1), chunkandpostag(2)
	chunkandpostag$(m,n)^*$	chunkandpostag(m) & chunkandpostag(n)	chunkandpostag$(-1,0)$, chunkandpostag$(0,1)$
BUILD	cons(n)	The head word, constituent (or POS) label, and start/join annotation of the nth tree. Start/join annotation omitted if $n \geq 0$.	cons(0), cons(-1), cons(-2), cons(1), cons(2)
	cons$(m,n)^*$	cons(m) & cons(n)	cons$(-1,0)$, cons$(0,1)$
	cons$(m,n,p)^\dagger$	cons(m), cons(n), & cons(p).	cons$(0,-1,-2)$, cons$(0,1,2)$, cons$(-1,0,1)$
	punctuation	The constituent we could join (1) contains a "[" and the current tree is a "]"; (2) contains a "," and the current tree is a ","; (3) spans the entire sentence and current tree is "."	bracketsmatch, iscomma, endofsentence
CHECK	checkcons$(n)^*$	The head word, constituent (or POS) label of the nth tree, and the label of proposed constituent. $begin$ and $last$ are first and last child (resp.) of proposed constituent.	checkcons$(last)$, checkcons$(begin)$
	checkcons$(m,n)^*$	checkcons(m) & checkcons(n)	checkcons$(i, last)$, $begin \leq i < last$
	production	Constituent label of parent (X), and constituent or POS labels of children $(X_1 \ldots X_n)$ of proposed constituent	production$=X \rightarrow X_1 \ldots X_n$
	surround$(n)^*$	POS tag and word of the nth leaf to the left of the constituent, if $n < 0$, or to the right of the constituent, if $n > 0$	surround(1), surround(2), surround(-1), surround(-2)

Table 3: Contextual Information Used by Probability Models (* = all backed-off contexts are used, † = only backed-off contexts that include head word of current tree, i.e., 0th tree, are used)

special estimation technique is required to combine features that encode different kinds of contextual predicates, like punctuation and cons$(0,1,2)$. As a result, experimenters need only worry about *what* features to use, and not *how* to use them.

We then use the models p_{tag}, p_{chunk}, p_{build}, and p_{check} to define a function score, which the search procedure uses to rank derivations of incomplete and complete parse trees. For each model, the corresponding conditional probability is defined as usual:

$$p(a|b) = \frac{p(a,b)}{\sum_{a' \in A} p(a',b)}$$

For notational convenience, define q as follows

$$q(a|b) = \begin{cases} p_{\text{tag}}(a|b) & \text{if } a \text{ is an action from TAG} \\ p_{\text{chunk}}(a|b) & \text{if } a \text{ is an action from CHUNK} \\ p_{\text{build}}(a|b) & \text{if } a \text{ is an action from BUILD} \\ p_{\text{check}}(a|b) & \text{if } a \text{ is an action from CHECK} \end{cases}$$

Let $\mathsf{deriv}(T) = \{a_1, \ldots, a_n\}$ be the derivation of a parse T, where T is not necessarily complete, and where each a_i is an action of some tree-building procedure. By design, the tree-building procedures guarantee that $\{a_1, \ldots, a_n\}$ is the only derivation for the parse T. Then the score of T is merely the product of the conditional probabilities of the individual actions in its derivation:

$$\mathsf{score}(T) = \prod_{a_i \in \mathsf{deriv}(T)} q(a_i|b_i)$$

where b_i is the context in which a_i was decided.

4 Search

The search heuristic attempts to find the best parse T^*, defined as:

$$T^* = \arg \max_{T \in \mathsf{trees}(S)} \mathsf{score}(T)$$

where $\mathsf{trees}(S)$ are all the complete parses for an input sentence S.

The heuristic employs a breadth-first search (BFS) which does not explore the entire frontier, but rather, explores only at most the top K scoring incomplete parses in the frontier, and terminates when it has found M complete parses, or when all the hypotheses have been exhausted. Furthermore, if $\{a_1 \ldots a_n\}$ are the possible actions for a given procedure on a derivation with context b, and they are sorted in decreasing order according to $q(a_i|b)$, we only consider exploring those actions $\{a_1 \ldots a_m\}$ that hold most of the probability mass, where m is defined as follows:

$$m = \max_m \sum_{i=1}^{m} q(a_i|b) < Q$$

and where Q is a threshold less than 1. The search also uses a *Tag Dictionary* constructed from training data, described in (Ratnaparkhi, 1996), that reduces the number of actions explored by the tagging model. Thus there are three parameters for the search heuristic, namely K,M, and Q and all experiments reported in this paper use $K = 20$, $M = 20$, and $Q = .95$[1] Table 4 describes the top K BFS and the semantics of the supporting functions.

It should be emphasized that if $K > 1$, the parser does not commit to a single POS or chunk assignment for the input sentence before building constituent structure. All three of the passes described in section 2 are integrated in the search, i.e., when parsing a test sentence, the input to the second pass consists of K of the best distinct POS tag assignments for the input sentence. Likewise, the input to the third pass consists of K of the best distinct chunk and POS tag assignments for the input sentence.

The top K BFS described above exploits the observed property that the individual steps of correct derivations tend to have high probabilities, and thus avoids searching a large fraction of the search space. Since, in practice, it only does a constant amount of work to advance each step in a derivation, and since derivation lengths are roughly proportional to the sentence length, we would expect it to run in linear observed time with respect to sentence length. Figure 8 confirms our assumptions about the linear observed running time.

5 Experiments

The maximum entropy parser was trained on sections 2 through 21 (roughly 40000 sentences) of the Penn Treebank Wall St. Journal corpus, release 2 (Marcus, Santorini, and Marcinkiewicz, 1994), and tested on section 23 (2416 sentences) for comparison with other work. All trees were stripped of their semantic tags (e.g., `-LOC`, `-BNF`, etc.), coreference information(e.g., `*-1`), and quotation marks (`` and '') for both training and testing. The PARSEVAL (Black and others, 1991) measures compare a proposed parse P with the corresponding correct treebank parse T as follows:

$$\text{Recall} = \frac{\#\text{ correct constituents in } P}{\#\text{ constituents in } T}$$

$$\text{Precision} = \frac{\#\text{ correct constituents in } P}{\#\text{ constituents in } P}$$

A constituent in P is "correct" if there exists a constituent in T of the same label that spans the same words. Table 5 shows results using the PARSEVAL measures, as well as results using the slightly more forgiving measures of (Collins, 1996) and (Magerman, 1995). Table 5 shows that the maximum entropy parser performs better than the parsers presented in (Collins, 1996) and (Magerman, 1995)[2],

[1] The parameters K,M, and Q were optimized on "held out" data separate from the training and test sets.

[2] Results for SPATTER on section 23 are reported in (Collins, 1996)

```
advance:     d × V ⟶ d_1 ... d_m    /* Applies relevant tree building procedure to d
                                    and returns list of new derivations whose action
                                    probabilities pass the threshold Q */
insert:      d × h ⟶ void           /* inserts d in heap h */
extract:     h ⟶ d                  /* removes and returns derivation in h
                                    with highest score */
completed:   d ⟶ {true,false}       /* returns true if and only if
                                    d is a complete derivation */

M = 20
K = 20
Q = .95
C = <empty heap>                    /* Heap of completed parses */
h_0 =<input sentence>               /* h_i contains derivations of length i */
while ( |C| < M )
      if ( ∀i, h_i is empty )
         then break
      i = max{i | h_i is non-empty}
      sz = min(K, |h_i|)
      for j = 1 to sz
          d_1 ... d_p = advance( extract(h_i), V )
          for q = 1 to p
              if (completed(d_q))
                 then insert(d_q, C)
                 else insert(d_q, h_{i+1})
```

Table 4: Top K BFS Search Heuristic

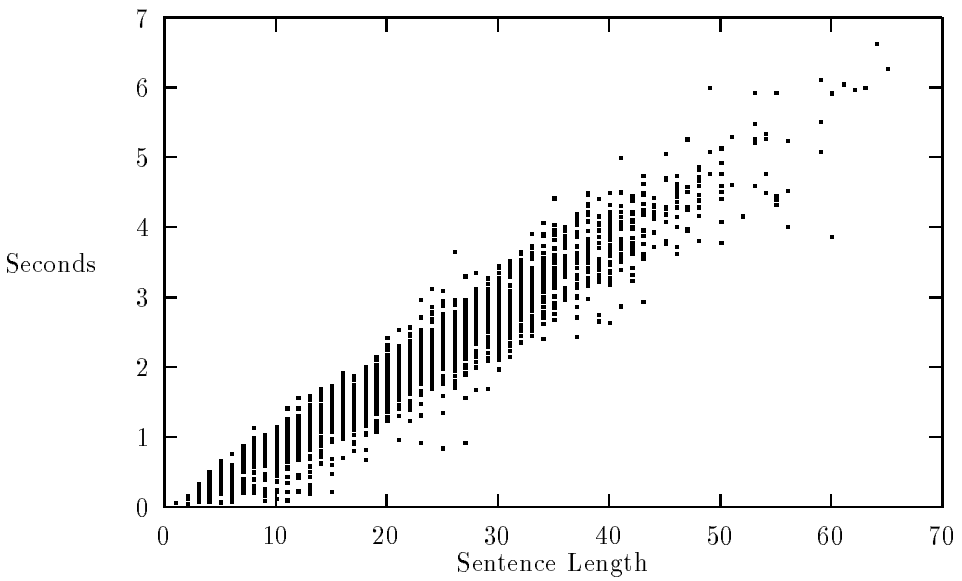

Figure 8: Observed running time of top K BFS on Section 23 of Penn Treebank WSJ, using one 167Mhz UltraSPARC processor and 256MB RAM of a Sun Ultra Enterprise 4000.

Parser	Precision	Recall
Maximum Entropy$^{\diamond}$	86.8%	85.6%
Maximum Entropy*	87.5%	86.3%
(Collins, 1996)*	85.7%	85.3%
(Magerman, 1995)*	84.3%	84.0%

Table 5: Results on 2416 sentences of section 23 (0 to 100 words in length) of the WSJ Treebank. Evaluations marked with $^{\diamond}$ ignore quotation marks. Evaluations marked with * collapse the distinction between `ADVP` and `PRT`, and ignore *all* punctuation.

which have the best previously published parsing accuracies on the Wall St. Journal domain.

It is often advantageous to produce the top N parses instead of just the top 1, since additional information can be used in a secondary model that reorders the top N and hopefully improves the quality of the top ranked parse. Suppose there exists a "perfect" reranking scheme that, for each sentence, magically picks the *best* parse from the top N parses produced by the maximum entropy parser, where the *best* parse has the highest average precision and recall when compared to the treebank parse. The performance of this "perfect" scheme is then an upper bound on the performance of any reranking scheme that might be used to reorder the top N parses. Figure 9 shows that the "perfect" scheme would achieve roughly 93% precision and recall, which is a dramatic increase over the top 1 accuracy of 87% precision and 86% recall. Figure 10 shows that the "Exact Match", which counts the percentage of times the proposed parse P is identical (excluding POS tags) to the treebank parse T, rises substantially to about 53% from 30% when the "perfect" scheme is applied. For this reason, research into reranking schemes appears to be a promising step towards the goal of improving parsing accuracy.

6 Comparison With Previous Work

The two parsers which have previously reported the best accuracies on the Penn Treebank Wall St. Journal are the bigram parser described in (Collins, 1996) and the SPATTER parser described in (Jelinek et al., 1994; Magerman, 1995). The parser presented here outperforms both the bigram parser and the SPATTER parser, and uses different modelling technology and different information to drive its decisions.

The bigram parser is a statistical CKY-style chart parser, which uses cooccurrence statistics of head-modifier pairs to find the best parse. The maximum entropy parser is a statistical shift-reduce style parser that cannot always access head-modifier pairs. For example, the checkcons(m, n) predicate of the maximum entropy parser may use two words such that *neither* is the intended head of the proposed consituent that the CHECK procedure must judge. And unlike the bigram parser, the maximum entropy parser cannot use head word information besides "flat" chunks in the right context.

The bigram parser uses a backed-off estimation scheme that is customized for a particular task, whereas the maximum entropy parser uses a general purpose modelling technique. This allows the maximum entropy parser to easily integrate varying kinds of features, such as those for punctuation, whereas the bigram parser uses hand-crafted punctuation rules. Furthermore, the customized estimation framework of the bigram parser must use information that has been carefully selected for its value, whereas the maximum entropy framework robustly integrates any kind of information, obviating the need to screen it first.

The SPATTER parser is a history-based parser that uses decision tree models to guide the operations of a few tree building procedures. It differs from the maximum entropy parser in how it builds trees and more critically, in how its decision trees use information. The SPATTER decision trees use predicates on word classes created with a statistical clustering technique, whereas the maximum entropy parser uses predicates that contain merely the words themselves, and thus lacks the need for a (typically expensive) word clustering procedure. Furthermore, the top K BFS search heuristic appears to be much simpler than the stack decoder algorithm outlined in (Magerman, 1995).

7 Conclusion

The maximum entropy parser presented here achieves a parsing accuracy which exceeds the best previously published results, and parses a test sentence in linear observed time, with respect to the sentence length. It uses simple and concisely specified predicates which can added or modified quickly with little human effort under the maximum entropy framework. Lastly, this paper clearly demonstrates that schemes for reranking the top 20 parses deserve research effort since they could yield vastly better accuracy results.

8 Acknowledgements

Many thanks to Mike Collins and Professor Mitch Marcus from the University of Pennsylvania for their helpful comments on this work.

References

Berger, Adam, Stephen A. Della Pietra, and Vincent J. Della Pietra. 1996. A Maximum Entropy Approach to Natural Language Processing. *Computational Linguistics*, 22(1):39–71.

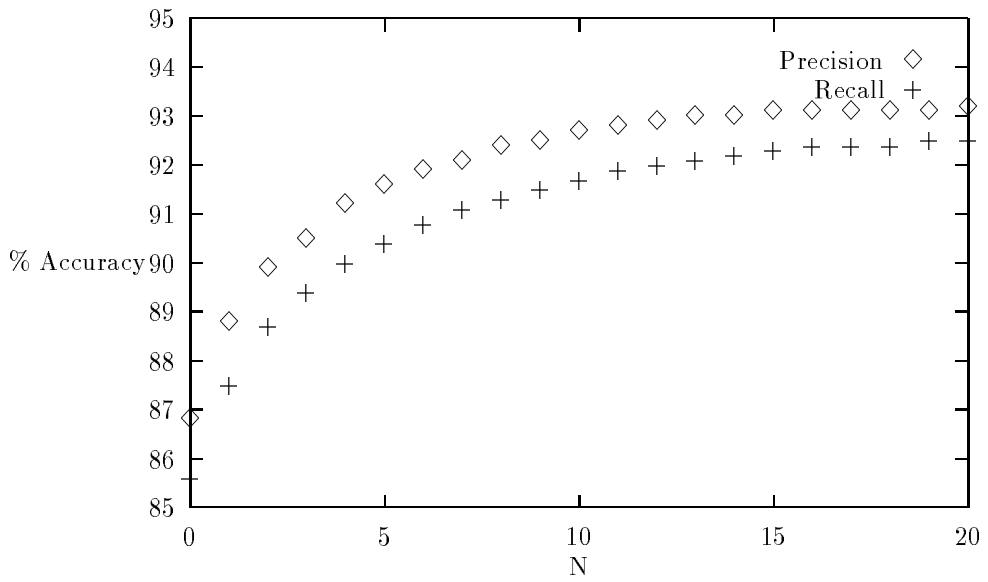

Figure 9: Precision & recall of a "perfect" reranking scheme for the top N parses of section 23 of the WSJ Treebank, as a function of N. Evaluation ignores quotation marks.

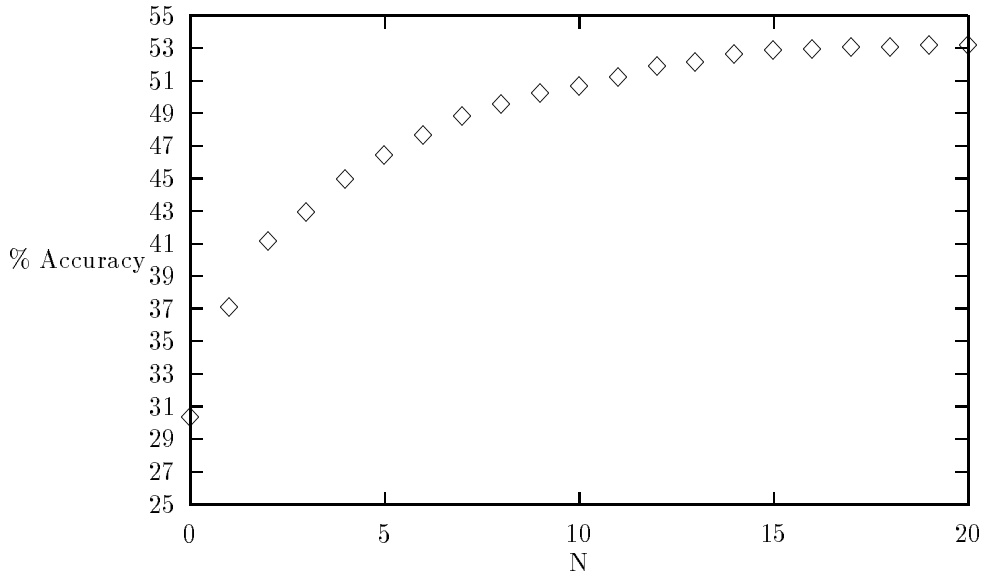

Figure 10: Exact match of a "perfect" reranking scheme for the top N parses of section 23 of the WSJ Treebank, as a function of N. Evaluation ignores quotation marks.

Black, Ezra et al. 1991. A Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars. In *Proceedings of the February 1991 DARPA Speech and Natural Language Workshop*, pages 306–311.

Black, Ezra, Fred Jelinek, John Lafferty, David M. Magerman, Robert Mercer, and Salim Roukos. 1993. Towards History-based Grammars: Using Richer Models for Probabilistic Parsing. In *Proceedings of the 31st Annual Meeting of the ACL*, Columbus, Ohio.

Collins, Michael John. 1996. A New Statistical Parser Based on Bigram Lexical Dependencies. In *Proceedings of the 34th Annual Meeting of the ACL*.

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

Jelinek, Fred, John Lafferty, David M. Magerman, Robert Mercer, Adwait Ratnaparkhi, and Salim Roukos. 1994. Decision Tree Parsing using a Hidden Derivational Model. In *Proceedings of the Human Language Technology Workshop*, pages 272–277. ARPA.

Lau, Ray, Ronald Rosenfeld, and Salim Roukos. 1993. Adaptive Language Modeling Using The Maximum Entropy Principle. In *Proceedings of the Human Language Technology Workshop*, pages 108–113. ARPA.

Magerman, David M. 1995. Statistical Decision-Tree Models for Parsing. In *Proceedings of the 33rd Annual Meeting of the ACL*.

Marcus, Mitchell P., Beatrice Santorini, and Mary Ann Marcinkiewicz. 1994. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Ratnaparkhi, Adwait. 1996. A Maximum Entropy Part of Speech Tagger. In *Conference on Empirical Methods in Natural Language Processing*, University of Pennsylvania, May 17–18.

Ratnaparkhi, Adwait. 1997. A Simple Introduction to Maximum Entropy Models for Natural Language Processing. Technical Report 97-08, Institute for Research in Cognitive Science, University of Pennsylvania.