Coarse-to-fine n-best parsing and MaxEnt discriminative reranking

Eugene Charniak and Mark Johnson
Brown Laboratory for Linguistic Information Processing (BLLIP)
Brown University
Providence, RI 02912
{mj|ec}@cs.brown.edu

Abstract

Discriminative reranking is one method for constructing high-performance statistical parsers (Collins, 2000). A discriminative reranker requires a source of candidate parses for each sentence. This paper describes a simple yet novel method for constructing sets of 50-best parses based on a coarse-to-fine generative parser (Charniak, 2000). This method generates 50-best lists that are of substantially higher quality than previously obtainable. We used these parses as the input to a MaxEnt reranker (Johnson et al., 1999; Riezler et al., 2002) that selects the best parse from the set of parses for each sentence, obtaining an f-score of 91.0% on sentences of length 100 or less.

1 Introduction

We describe a reranking parser which uses a regularized MaxEnt reranker to select the best parse from the 50-best parses returned by a generative parsing model. The 50-best parser is a probabilistic parser that on its own produces high quality parses; the maximum probability parse trees (according to the parser's model) have an f-score of 0.897 on section 23 of the Penn Treebank (Charniak, 2000), which is still state-of-the-art. However, the 50 best (i.e., the 50 highest probability) parses of a sentence often contain considerably better parses (in terms of f-score); this paper describes a 50-best parsing algorithm with an oracle f-score of 96.8 on the same data.

The reranker attempts to select the best parse for a sentence from the 50-best list of possible parses for the sentence. Because the reranker only has to consider a relatively small number of parses per sentences, it is not necessary to use dynamic programming, which permits the features to be essentially arbitrary functions of the parse trees. While our reranker does not achieve anything like the oracle f-score, the parses it selects do have an f-score of 91.0, which is considerably better than the maximum probability parses of the n-best parser.

In more detail, for each string s the n-best parsing algorithm described in section 2 returns the n highest probability parses $\mathcal{Y}(s) = \{y_1(s), \ldots, y_n(s)\}$ together with the probability $p(y)$ of each parse y according to the parser's probability model. The number n of parses was set to 50 for the experiments described here, but some simple sentences actually received fewer than 50 parses (so n is actually a function of s). Each yield or terminal string in the training, development and test data sets is mapped to such an n-best list of parse/probability pairs; the cross-validation scheme described in Collins (2000) was used to avoid training the n-best parser on the sentence it was being used to parse.

A feature extractor, described in section 3, is a vector of m functions $f = (f_1, \ldots, f_m)$, where each f_j maps a parse y to a real number $f_j(y)$, which is the value of the jth feature on y. So a feature extractor maps each y to a vector of feature values $f(y) = (f_1(y), \ldots, f_m(y))$.

Our reranking parser associates a parse with a

Proceedings of the 43rd Annual Meeting of the ACL, pages 173–180,
Ann Arbor, June 2005.

score $v_\theta(y)$, which is a linear function of the feature values $f(y)$. That is, each feature f_j is associated with a weight θ_j, and the feature values and weights define the score $v_\theta(y)$ of each parse y as follows:

$$v_\theta(y) \;=\; \theta \cdot f(y) = \sum_{j=1}^{m} \theta_j f_j(y).$$

Given a string s, the reranking parser's output $\hat{y}(s)$ on string s is the highest scoring parse in the n-best parses $\mathcal{Y}(s)$ for s, i.e.,

$$\hat{y}(s) \;=\; \arg\max_{y \in \mathcal{Y}(s)} v_\theta(y).$$

The feature weight vector θ is estimated from the labelled training corpus as described in section 4. Because we use labelled training data we know the correct parse $y_\star(s)$ for each sentence s in the training data. The correct parse $y_\star(s)$ is not always a member of the n-best parser's output $\mathcal{Y}(s)$, but we can identify the parses $\mathcal{Y}_+(s)$ in $\mathcal{Y}(s)$ with the highest f-scores. Informally, the estimation procedure finds a weight vector θ that maximizes the score $v_\theta(y)$ of the parses $y \in \mathcal{Y}_+(s)$ relative to the scores of the other parses in $\mathcal{Y}(s)$, for each s in the training data.

2 Recovering the *n*-best parses using coarse-to-fine parsing

The major difficulty in n-best parsing, compared to 1-best parsing, is dynamic programming. For example, n-best parsing is straight-forward in best-first search or beam search approaches that do not use dynamic programming: to generate more than one parse, one simply allows the search mechanism to create successive versions to one's heart's content.

A good example of this is the Roark parser (Roark, 2001) which works left-to right through the sentence, and abjures dynamic programming in favor of a beam search, keeping some large number of possibilities to extend by adding the next word, and then re-pruning. At the end one has a beam-width's number of best parses (Roark, 2001).

The Collins parser (Collins, 1997) *does* use dynamic programming in its search. That is, whenever a constituent with the same history is generated a second time, it is discarded if its probability is lower than the original version. If the opposite is true, then the original is discarded. This is fine if one only wants the first-best, but obviously it does not directly enumerate the n-best parses.

However, Collins (Collins, 2000; Collins and Koo, in submission) has created an n-best version of his parser by turning off dynamic programming (see the user's guide to Bikel's re-implementation of Collins' parser, http://www.cis.upenn.edu/ dbikel/software.html#stat-parser). As with Roark's parser, it is necessary to add a beam-width constraint to make the search tractable. With a beam width of 1000 the parser returns something like a 50-best list (Collins, personal communication), but the actual number of parses returned for each sentences varies. However, turning off dynamic programming results in a loss in efficiency. Indeed, Collins's n-best list of parses for section 24 of the Penn tree-bank has some sentences with only a single parse, because the n-best parser could not find any parses.

Now there are two known ways to produce n-best parses while retaining the use of dynamic programming: the obvious way and the clever way.

The clever way is based upon an algorithm developed by Schwartz and Chow (1990). Recall the key insight in the Viterbi algorithm: in the optimal parse the parsing decisions at each of the choice points that determine a parse must be optimal, since otherwise one could find a better parse. This insight extends to n-best parsing as follows. Consider the second-best parse: if it is to differ from the best parse, then at least one of its parsing decisions must be suboptimal. In fact, all but one of the parsing decisions in second-best parse must be optimal, and the one suboptimal decision must be the second-best choice at that choice point. Further, the nth-best parse can only involve at most n suboptimal parsing decisions, and all but one of these must be involved in one of the second through the $n-1$th-best parses. Thus the basic idea behind this approach to n-best parsing is to first find the best parse, then find the second-best parse, then the third-best, and so on. The algorithm was originally described for hidden Markov models.

Since this first draft of this paper we have become aware of two PCFG implementations of this algorithm (Jimenez and Marzal, 2000; Huang and Chang, 2005). The first was tried on relatively small grammars, while the second was implemented on top of the Bikel re-implementation of the Collins

parser (Bikel, 2004) and achieved oracle results for 50-best parses similar to those we report below.

Here, however, we describe how to find n-best parses in a more straight-forward fashion. Rather than storing a single best parse of each edge, one stores n of them. That is, when using dynamic programming, rather than throwing away a candidate if it scores less than the best, one keeps it if it is one of the top n analyses for this edge discovered so far. This is really very straight-forward. The problem is space. Dynamic programming parsing algorithms for PCFGs require $O(m^2)$ dynamic programming states, where m is the length of the sentence, so an n-best parsing algorithm requires $O(nm^2)$. However things get much worse when the grammar is bilexicalized. As shown by Eisner (Eisner and Satta, 1999) the dynamic programming algorithms for bilexicalized PCFGs require $O(m^3)$ states, so a n-best parser would require $O(nm^3)$ states. Things become worse still in a parser like the one described in Charniak (2000) because it conditions on (and hence splits the dynamic programming states according to) features of the grandparent node in addition to the parent, thus multiplying the number of possible dynamic programming states even more. Thus nobody has implemented this version.

There is, however, one particular feature of the Charniak parser that mitigates the space problem: it is a "coarse-to-fine" parser. By "coarse-to-fine" we mean that it first produces a crude version of the parse using coarse-grained dynamic programming states, and then builds fine-grained analyses by splitting the most promising of coarse-grained states.

A prime example of this idea is from Goodman (1997), who describes a method for producing a simple but crude approximate grammar of a standard context-free grammar. He parses a sentence using the approximate grammar, and the results are used to constrain the search for a parse with the full CFG. He finds that total parsing time is greatly reduced.

A somewhat different take on this paradigm is seen in the parser we use in this paper. Here the parser first creates a parse forest based upon a much less complex version of the complete grammar. In particular, it only looks at standard CFG features, the parent and neighbor labels. Because this grammar encodes relatively little state information, its dynamic programming states are relatively coarse and hence there are comparatively few of them, so it can be efficiently parsed using a standard dynamic programming bottom-up CFG parser. However, precisely because this first stage uses a grammar that ignores many important contextual features, the best parse it finds will not, in general, be the best parse according to the finer-grained second-stage grammar, so clearly we do not want to perform best-first parsing with this grammar. Instead, the output of the first stage is a polynomial-sized packed parse forest which records the left and right string positions for each local tree in the parses generated by this grammar. The edges in the packed parse forest are then pruned, to focus attention on the coarse-grained states that are likely to correspond to high-probability fine-grained states. The edges are then pruned according to their marginal probability conditioned on the string s being parsed as follows:

$$p(n^i_{j,k} \mid s) = \frac{\alpha(n^i_{j,k})\beta(n^i_{j,k})}{p(s)} \quad (1)$$

Here $n^i_{j,k}$ is a constituent of type i spanning the words from j to k, $\alpha(n^i_{j,k})$ is the outside probability of this constituent, and $\beta(n^i_{j,k})$ is its inside probability. From parse forest both α and β can be computed in time proportional to the size of the compact forest. The parser then removes all constituents $n^i_{j,k}$ whose probability falls below some preset threshold. In the version of this parser available on the web, this threshold is on the order of 10^{-4}.

The unpruned edges are then exhaustively evaluated according to the fine-grained probabilistic model; in effect, each coarse-grained dynamic programming state is split into one or more fine-grained dynamic programming states. As noted above, the fine-grained model conditions on information that is not available in the coarse-grained model. This includes the lexical head of one's parents, the part of speech of this head, the parent's and grandparent's category labels, etc. The fine-grained states investigated by the parser are constrained to be refinements of the coarse-grained states, which drastically reduces the number of fine-grained states that need to be investigated.

It is certainly possible to do dynamic programming parsing directly with the fine-grained grammar, but precisely because the fine-grained grammar

conditions on a wide variety of non-local contextual information there would be a very large number of different dynamic programming states, so direct dynamic programming parsing with the fine-grained grammar would be very expensive in terms of time and memory.

As the second stage parse evaluates all the remaining constituents in all of the contexts in which they appear (e.g., what are the possible grand-parent labels) it keeps track of the most probable expansion of the constituent in that context, and at the end is able to start at the root and piece together the overall best parse.

Now comes the easy part. To create a 50-best parser we simply change the fine-grained version of 1-best algorithm in accordance with the "obvious" scheme outlined earlier in this section. The first, coarse-grained, pass is not changed, but the second, fine-grained, pass keeps the n-best possibilities at each dynamic programming state, rather than keeping just first best. When combining two constituents to form a larger constituent, we keep the best 50 of the 2500 possibilities they offer. Naturally, if we keep each 50-best list sorted, we do nothing like 2500 operations.

The experimental question is whether, in practice, the coarse-to-fine architecture keeps the number of dynamic programming states sufficiently low that space considerations do not defeat us.

The answer seems to be yes. We ran the algorithm on section 24 of the Penn WSJ tree-bank using the default pruning settings mentioned above. Table 1 shows how the number of fine-grained dynamic programming states increases as a function of sentence length for the sentences in section 24 of the Treebank. There are no sentences of length greater than 69 in this section. Columns two to four show the number of sentences in each bucket, their average length, and the average number of fine-grained dynamic programming structures per sentence. The final column gives the value of the function $100 * L^{1.5}$ where L is the average length of sentences in the bucket. Except for bucket 6, which is abnormally low, it seems that this add-hoc function tracks the number of structures quite well. Thus the number of dynamic programming states does not grow as L^2, much less as L^3.

To put the number of these structures per sentence in perspective, consider the size of such structures. Each one must contain a probability, the non-terminal label of the structure, and a vector of pointers to it's children (an average parent has slightly more than two children). If one were concerned about every byte this could be made quite small. In our implementation probably the biggest factor is the STL overhead on vectors. If we figure we are using, say, 25 bytes per structure, the total space required is only 1.25Mb even for 50,000 dynamic programming states, so it is clearly not worth worrying about the memory required.

Len	Num sents	Av sen length	Av strs per sent	$100 * L^{1.5}$
0–9	225	6.04	1167	1484
10–19	725	15.0	4246	5808
20–29	795	24.2	9357	11974
30–39	465	33.8	15893	19654
40–49	162	43.2	21015	28440
50–59	35	52.8	30670	38366
60–69	9	62.8	23405	49740

Table 1: Number of structures created as a function of sentence length

n	1	2	10	25	50
f-score	0.897	0.914	0.948	0.960	0.968

Table 2: Oracle f-score as a function of number n of n-best parses

The resulting n-bests are quite good, as shown in Table 2. (The results are for all sentences of section 23 of the WSJ tree-bank of length ≤ 100.) From the 1-best result we see that the base accuracy of the parser is 89.7%.[1] 2-best and 10-best show dramatic oracle-rate improvements. After that things start to slow down, and we achieve an oracle rate of 0.968 at 50-best. To put this in perspective, Roark (Roark, 2001) reports oracle results of 0.941 (with the same experimental setup) using his parser to return a variable number of parses. For the case cited his parser returns, on average, 70 parses per sentence.

Finally, we note that 50-best parsing is only a fac-

[1] Charniak in (Charniak, 2000) cites an accuracy of 89.5%. Fixing a few very small bugs discovered by users of the parser accounts for the difference.

tor of two or three slower than 1-best.

3 Features for reranking parses

This section describes how each parse y is mapped to a feature vector $f(y) = (f_1(y), \ldots, f_m(y))$. Each feature f_j is a function that maps a parse to a real number. The first feature $f_1(y) = \log p(y)$ is the logarithm of the parse probability p according to the n-best parser model. The other features are integer valued; informally, each feature is associated with a particular configuration, and the feature's value $f_j(y)$ is the number of times that the configuration that f_j indicates. For example, the feature $f_{\text{eat pizza}}(y)$ counts the number of times that a phrase in y headed by *eat* has a complement phrase headed by *pizza*.

Features belong to feature schema, which are abstract schema from which specific features are instantiated. For example, the feature $f_{\text{eat pizza}}$ is an instance of the "Heads" schema. Feature schema are often parameterized in various ways. For example, the "Heads" schema is parameterized by the type of heads that the feature schema identifies. Following Grimshaw (1997), we associate each phrase with a lexical head and a function head. For example, the lexical head of an NP is a noun while the functional head of an NP is a determiner, and the lexical head of a VP is a main verb while the functional head of VP is an auxiliary verb.

We experimented with various kinds of feature selection, and found that a simple count threshold performs as well as any of the methods we tried. Specifically, we ignored all features that did not vary on the parses of at least t sentences, where t is the count threshold. In the experiments described below $t = 5$, though we also experimented with $t = 2$.

The rest of this section outlines the feature schemata used in the experiments below. These feature schemata used here were developed using the n-best parses provided to us by Michael Collins approximately a year before the n-best parser described here was developed. We used the division into preliminary training and preliminary development data sets described in Collins (2000) while experimenting with feature schemata; i.e., the first 36,000 sentences of sections 2–20 were used as preliminary training data, and the remaining sentences of sections 20 and 21 were used as preliminary development data. It is worth noting that developing feature schemata is much more of an art than a science, as adding or deleting a single schema usually does not have a significant effect on performance, yet the overall impact of many well-chosen schemata can be dramatic.

Using the 50-best parser output described here, there are 1,148,697 features that meet the count threshold of at least 5 on the main training data (i.e., Penn treebank sections 2–21). We list each feature schema's name, followed by the number of features in that schema with a count of at least 5, together with a brief description of the instances of the schema and the schema's parameters.

CoPar (10) The instances of this schema indicate conjunct parallelism at various different depths. For example, conjuncts which have the same label are parallel at depth 0, conjuncts with the same label and whose children have the same label are parallel at depth 1, etc.

CoLenPar (22) The instances of this schema indicate the binned difference in length (in terms of number of preterminals dominated) in adjacent conjuncts in the same coordinated structures, conjoined with a boolean flag that indicates whether the pair is final in the coordinated phrase.

RightBranch (2) This schema enables the reranker to prefer right-branching trees. One instance of this schema returns the number of nonterminal nodes that lie on the path from the root node to the right-most non-punctuation preterminal node, and the other instance of this schema counts the number of the other nonterminal nodes in the parse tree.

Heavy (1049) This schema classifies nodes by their category, their binned length (i.e., the number of preterminals they dominate), whether they are at the end of the sentence and whether they are followed by punctuation.

Neighbours (38,245) This schema classifies nodes by their category, their binned length, and the part of speech categories of the ℓ_1 preterminals to the node's left and the ℓ_2 preterminals to the

node's right. ℓ_1 and ℓ_2 are parameters of this schema; here $\ell_1 = 1$ or $\ell_1 = 2$ and $\ell_2 = 1$.

Rule (271,655) The instances of this schema are local trees, annotated with varying amounts of contextual information controlled by the schema's parameters. This schema was inspired by a similar schema in Collins and Koo (in submission). The parameters to this schema control whether nodes are annotated with their preterminal heads, their terminal heads and their ancestors' categories. An additional parameter controls whether the feature is specialized to embedded or non-embedded clauses, which roughly corresponds to Emonds' "non-root" and "root" contexts (Emonds, 1976).

NGram (54,567) The instances of this schema are ℓ-tuples of adjacent children nodes of the same parent. This schema was inspired by a similar schema in Collins and Koo (in submission). This schema has the same parameters as the Rule schema, plus the length ℓ of the tuples of children ($\ell = 2$ here).

Heads (208,599) The instances of this schema are tuples of head-to-head dependencies, as mentioned above. The category of the node that is the least common ancestor of the head and the dependent is included in the instance (this provides a crude distinction between different classes of arguments). The parameters of this schema are whether the heads involved are lexical or functional heads, the number of heads in an instance, and whether the lexical item or just the head's part of speech are included in the instance.

LexFunHeads (2,299) The instances of this feature are the pairs of parts of speech of the lexical head and the functional head of nodes in parse trees.

WProj (158,771) The instances of this schema are preterminals together with the categories of ℓ of their closest maximal projection ancestors. The parameters of this schema control the number ℓ of maximal projections, and whether the preterminals and the ancestors are lexicalized.

Word (49,097) The instances of this schema are lexical items together with the categories of ℓ of their immediate ancestor nodes, where ℓ is a schema parameter ($\ell = 2$ or $\ell = 3$ here). This feature was inspired by a similar feature in Klein and Manning (2003).

HeadTree (72,171) The instances of this schema are tree fragments consisting of the local trees consisting of the projections of a preterminal node and the siblings of such projections. This schema is parameterized by the head type (lexical or functional) used to determine the projections of a preterminal, and whether the head preterminal is lexicalized.

NGramTree (291,909) The instances of this schema are subtrees rooted in the least common ancestor of ℓ contiguous preterminal nodes. This schema is parameterized by the number ℓ of contiguous preterminals ($\ell = 2$ or $\ell = 3$ here) and whether these preterminals are lexicalized.

4 Estimating feature weights

This section explains how we estimate the feature weights $\theta = (\theta_1, \ldots, \theta_m)$ for the feature functions $f = (f_1, \ldots, f_m)$. We use a MaxEnt estimator to find the feature weights $\hat{\theta}$, where L is the loss function and R is a regularization penalty term:

$$\hat{\theta} = \arg\min_{\theta} L_D(\theta) + R(\theta).$$

The training data $D = (s_1, \ldots, s_{n'})$ is a sequence of sentences and their correct parses $y_\star(s_1), \ldots, y_\star(s_n)$. We used the 20-fold cross-validation technique described in Collins (2000) to compute the n-best parses $\mathcal{Y}(s)$ for each sentence s in D. In general the correct parse $y_\star(s)$ is not a member of $\mathcal{Y}(s)$, so instead we train the reranker to identify one of the best parses $\mathcal{Y}_+(s) = \arg\max_{y \in \mathcal{Y}(s)} F_{y_\star(s)}(y)$ in the n-best parser's output, where $F_{y_\star}(y)$ is the Parseval f-score of y evaluated with respect to $y_\star$.

Because there may not be a unique best parse for each sentence (i.e., $|\mathcal{Y}_+(s)| > 1$ for some sentences s) we used the variant of MaxEnt described in Riezler et al. (2002) for partially labelled training data.

Recall the standard MaxEnt conditional probability model for a parse $y \in \mathcal{Y}$:

$$\begin{aligned} \mathrm{P}_\theta(y|\mathcal{Y}) &= \frac{\exp v_\theta(y)}{\sum_{y' \in \mathcal{Y}} \exp v_\theta(y')}, \text{where} \\ v_\theta(y) &= \theta \cdot f(y) = \sum_{j=1}^{m} \theta_j f_j(y). \end{aligned}$$

The loss function L_D proposed in Riezler et al. (2002) is just the negative log conditional likelihood of the best parses $\mathcal{Y}_+(s)$ relative to the n-best parser output $\mathcal{Y}(s)$:

$$\begin{aligned} L_D(\theta) &= -\sum_{i=1}^{n'} \log \mathrm{P}_\theta(\mathcal{Y}_+(s_i)|\mathcal{Y}(s_i)), \text{where} \\ \mathrm{P}_\theta(\mathcal{Y}_+|\mathcal{Y}) &= \sum_{y \in \mathcal{Y}_+} \mathrm{P}_\theta(y|\mathcal{Y}) \end{aligned}$$

The partial derivatives of this loss function, which are required by the numerical estimation procedure, are:

$$\begin{aligned} \frac{\partial L_D}{\theta_j} &= \sum_{i=1}^{n'} \mathrm{E}_\theta[f_j|\mathcal{Y}(s_i)] - E_\theta[f_j|\mathcal{Y}_+(s_i)] \\ \mathrm{E}_\theta[f|\mathcal{Y}] &= \sum_{y \in \mathcal{Y}} f(y) \mathrm{P}_\theta(y|\mathcal{Y}) \end{aligned}$$

In the experiments reported here, we used a Gaussian or quadratic regularizer $R(w) = c \sum_{j=1}^{m} w_j^2$, where c is an adjustable parameter that controls the amount of regularization, chosen to optimize the reranker's f-score on the development set (section 24 of the treebank).

We used the Limited Memory Variable Metric optimization algorithm from the PETSc/TAO optimization toolkit (Benson et al., 2004) to find the optimal feature weights $\hat{\theta}$ because this method seems substantially faster than comparable methods (Malouf, 2002). The PETSc/TAO toolkit provides a variety of other optimization algorithms and flags for controlling convergence, but preliminary experiments on the Collins' trees with different algorithms and early stopping did not show any performance improvements, so we used the default PETSc/TAO setting for our experiments here.

5 Experimental results

We evaluated the performance of our reranking parser using the standard PARSEVAL metrics. We trained the n-best parser on sections 2–21 of the Penn Treebank, and used section 24 as development data to tune the mixing parameters of the smoothing model. Similarly, we trained the feature weights θ with the MaxEnt reranker on sections 2–21, and adjusted the regularizer constant c to maximize the f-score on section 24 of the treebank. We did this both on the trees supplied to us by Michael Collins, and on the output of the n-best parser described in this paper. The results are presented in Table 3. The n-best parser's most probable parses are already of state-of-the-art quality, but the reranker further improves the f-score.

n-best trees	**f-score**
New	0.9102
Collins	0.9037

Table 3: Results on new n-best trees and Collins n-best trees, with weights estimated from sections 2–21 and the regularizer constant c adjusted for optimal f-score on section 24 and evaluated on sentences of length less than 100 in section 23.

6 Conclusion

This paper has described a dynamic programming n-best parsing algorithm that utilizes a heuristic coarse-to-fine refinement of parses. Because the coarse-to-fine approach prunes the set of possible parse edges beforehand, a simple approach which enumerates the n-best analyses of each parse edge is not only practical but quite efficient.

We use the 50-best parses produced by this algorithm as input to a MaxEnt discriminative reranker. The reranker selects the best parse from this set of parses using a wide variety of features. The system we described here has an f-score of 0.91 when trained and tested using the standard PARSEVAL framework.

This result is only slightly higher than the highest reported result for this test-set, Bod's (.907) (Bod, 2003). More to the point, however, is that the system we describe is reasonably efficient so it can be used for the kind of routine parsing currently being handled by the Charniak or Collins parsers. A 91.0 f-score represents a 13% reduction in f-

measure error over the best of these parsers.[2] Both the 50-best parser, and the reranking parser can be found at ftp://ftp.cs.brown.edu/pub/nlparser/, named parser and reranker respectively.

Acknowledgements We would like to thanks Michael Collins for the use of his data and many helpful comments, and Liang Huang for providing an early draft of his paper and very useful comments on our paper. Finally thanks to the National Science Foundation for its support (NSF IIS-0112432, NSF 9721276, and NSF DMS-0074276).

References

Steve Benson, Lois Curfman McInnes, Jorge J. Mor, and Jason Sarich. 2004. Tao users manual. Technical Report ANL/MCS-TM-242-Revision 1.6, Argonne National Laboratory.

Daniel M. Bikel. 2004. Intricacies of collins parsing model. *Computational Linguistics*, 30(4).

Rens Bod. 2003. An efficient implementation of an new dop model. In *Proceedings of the European Chapter of the Association for Computational Linguists*.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *The Proceedings of the North American Chapter of the Association for Computational Linguistics*, pages 132–139.

Michael Collins and Terry Koo. in submission. Discriminative reranking for natural language parsing. Technical report, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *The Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, San Francisco. Morgan Kaufmann.

Michael Collins. 2000. Discriminative reranking for natural language parsing. In *Machine Learning: Proceedings of the Seventeenth International Conference (ICML 2000)*, pages 175–182, Stanford, California.

Jason Eisner and Giorgio Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 457–464.

Joseph Emonds. 1976. *A Transformational Approach to English Syntax: Root, Structure-Preserving and Local Transformations.* Academic Press, New York, NY.

Joshua Goodman. 1997. Global thresholding and multiple-pass parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 1997)*.

Jane Grimshaw. 1997. Projection, heads, and optimality. *Linguistic Inquiry*, 28(3):373–422.

Liang Huang and David Chang. 2005. Better k-best parsing. Technical Report MS-CIS-05-08, Department of Computer Science, University of Pennsylvania.

Victor M. Jimenez and Andres Marzal. 2000. Computation of the n best parse trees for weighted and stochastic context-free grammars. In *Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition*. Springer LNCS 1876.

Mark Johnson, Stuart Geman, Stephen Canon, Zhiyi Chi, and Stefan Riezler. 1999. Estimators for stochastic "unification-based" grammars. In *The Proceedings of the 37th Annual Conference of the Association for Computational Linguistics*, pages 535–541, San Francisco. Morgan Kaufmann.

Dan Klein and Christopher Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*.

Robert Malouf. 2002. A comparison of algorithms for maximum entropy parameter estimation. In *Proceedings of the Sixth Conference on Natural Language Learning (CoNLL-2002)*, pages 49–55.

Stefan Riezler, Tracy H. King, Ronald M. Kaplan, Richard Crouch, John T. III Maxwell, and Mark Johnson. 2002. Parsing the wall street journal using a lexical-functional grammar and discriminative estimation techniques. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 271–278. Morgan Kaufmann.

Brian Roark. 2001. Probabilistic top-down parsing and language modeling. *Computational Linguistics*, 27(2):249–276.

R. Schwartz and Y.L. Chow. 1990. The n-best algorithm: An efficient and exact procedure for finding the n most likely sentence hypotheses. In *Proceedings of the IEEE International Conference on Acoustic, Speech, and Signal, Processing*, pages 81–84.

[2] This probably underestimates the actual improvement. There are no currently accepted figures for inter-annotater agreement on Penn WSJ, but it is no doubt well short of 100%. If we take 97% as a reasonable estimate of the the upper bound on tree-bank accuracy, we are instead talking about an 18% error reduction.