Induction of Greedy Controllers for Deterministic Treebank Parsers

Tom Kalt
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-9264
kalt@cs.umass.edu

Abstract

Most statistical parsers have used the grammar induction approach, in which a stochastic grammar is induced from a treebank. An alternative approach is to induce a controller for a given parsing automaton. Such controllers may be stochastic; here, we focus on greedy controllers, which result in deterministic parsers. We use decision trees to learn the controllers. The resulting parsers are surprisingly accurate and robust, considering their speed and simplicity. They are almost as fast as current part-of-speech taggers, and considerably more accurate than a basic unlexicalized PCFG parser. We also describe Markov parsing models, a general framework for parser modeling and control, of which the parsers reported here are a special case.

1 Introduction

A fundamental result of formal language theory is that the languages defined by context-free grammars are the same as those accepted by push-down automata. This result was recently extended to the stochastic case (Abney, et al., 1999). There are thus two main approaches to training a statistical parser: inducing stochastic grammars and inducing stochastic automata. Most recent work has employed grammar induction (Collins, 1999; Charniak, 2000). Examples of the automaton-induction approach are Hermjakob (1997), which described a deterministic parser, and Ratnaparkhi (1998), which described a stochastic parser.

The deterministic parsers reported in this paper are greedy versions of stochastic parsers based on Markov parsing models, described in section 3.3. A greedy parser takes the single most probable action at every choice point. It thus does the minimum amount of search possible. There will always be a tradeoff between speed on the one hand and accuracy and robustness on the other. Our aim, in studying greedy parsers, is to find out what levels of coverage and accuracy can be attained at the high-speed extreme of this tradeoff. There is no guarantee that a greedy parser will find the best parse, or indeed any complete parse. So the accuracy and coverage of greedy parsers are both interesting empirical questions. We find that they are almost as fast as current part-of-speech taggers, and they outperform basic unlexicalized PCFG parsers. While coverage is a concern, it is quite high (over 99%) for some of our parsers.

2 Previous work

Markov parsing models are an example of the history-based parsing approach (Black, et al., 1992). History-based parsing, broadly interpreted, includes most statistical parsers. Markov parsing models take a more automaton-oriented (or control-oriented) view of what history means, compared to the more grammar-oriented view of the original paper and most subsequent work.

Hermjakob (1997) described a deterministic shift-reduce parser. The control is learned by a hybrid of decision trees and decision lists. This work also used a rich hand-crafted semantic ontology. The state representation contained over 200 features, although it still worked rather well when this was reduced to 12. A notable feature was that good performance was achieved with very little training data (256 sentences). His results are not directly comparable with most other experiments, for several reasons, including the use of a subset of the Wall St. Journal corpus that used a closed lexicon of 3000 words.

Ratnaparkhi (1999) used a maximum entropy model to compute action probabilities for a bottom-up parser. His score function is an instance of a Markov parsing model, as defined in this paper (although he did not interpret his score as a probability). His parser performed at a level very close to state-of-the-art. His approach was similar to ours in a number of ways.

He used a beam search to find multiple parses.

Wong and Wu (1999) implemented a deterministic shift-reduce parser, using a novel variation on shift-reduce which employed a separate chunker-like phase as well as a base NP recognizer. They used decision trees to learn control. Their state representation was restricted to unlexicalized syntactic information.

The approach described here combines many elements which have previously been found useful. Left-corner parsing is discussed in Manning and Carpenter (1997) and Roark (2001). For search, Roark used beam search with non-backtracking top down automata, rather than the more usual chart-based search. Magerman (1994) used a parsing model based on decision tree techniques. Numerous papers (Manning and Carpenter, 1997; Johnson, 1998; Charniak et al., 1998; Roark, 2001) report that treebank binarization is advantageous.

3 Automaton Induction

3.1 General Concepts

Our approach to automaton induction is to view parsing as a control problem. The parsing automaton is a discrete dynamical system which we want to learn to control. At any given time, a parser is in some *state*. We use the term *state* as in control theory, and not as in automata theory. The state consists of the contents of the parser's data structures: a buffer holding the input sentence, a push-down stack, and a set of parse tree fragments, consisting of current or previous stack items connected by arcs. A parser has a finite set of *actions*, which perform simple operations on its data structures (pushing and popping the stack, dequeuing from the input buffer, and creating arcs between stack items). Performing an action changes the state. Most important, the parser has a *controller*, which chooses an action at each time step based on the current state. A map from states to actions is called a *policy*. If this map is a function, then the policy is deterministic, and results in a deterministic parser. If the map is a state-conditional distribution over actions, then the policy is stochastic, resulting in a stochastic parser. A deterministic parser returns a single parse for a given input, while a stochastic parser may return more than one. Thus the problem of inducing a stochastic parsing automaton consists in specifying the automaton's data structures, its dynamics (what the actions do), and then learning a stochastic policy.

In this paper, we assume that the parsing automaton always halts and outputs a tree. We can easily modify any parser so that this is the case. A parser fails when it is asked to perform an action that is impossible (e.g. shifting when there is no more input). When this happens, the parser terminates by creating a root node labeled FAIL, whose children are all complete constituents constructed so far, along with any unused input. Parsers can also fail by going into a cycle. This must be detected, which can be done by limiting the number of actions to some multiple of the input length. In practice, we have found that cycles are rare, and not difficult to handle. We also assume that the parsing automaton is *reversible*; that is, for a given input string, there is a one-to-one correspondence between parse trees and the sequence of actions that produces that tree. Because of reversibility, given a parser and a tree, we can easily determine the unique sequence of actions that the parser would use to produce that tree; we call this *unparsing*.

3.2 Deterministic Control

The parsers reported in this paper used deterministic controllers created as described below. Induction of deterministic control for a given parser, using a treebank, is a straightforward classification problem. We use the parser and the treebank to create a training set of *(state, action)* pairs, and we induce a function from states to actions, which is the deterministic controller. To create training instances from a treebank, we first unparse each tree to get an action sequence. We then use these actions to control the parser as it processes the corresponding input. At each time step, we create a training instance, consisting of the parser's current state and the action it takes from that state.

State: We said above that the parser's state consists of the contents of its data structures; we will call this the *complete* state. The state space of the complete state is infinite, as states include the contents of unbounded stacks and input buffers. We need to map this into a manageable number of equivalence classes. This is done in two stages. First, we restrict attention to a finite part of the complete state. That is, we map particular elements of the parser's data structures onto a feature vector. We refer to this as the state representation, and the choice of representation is a critical element of parser design. All the work reported here uses twelve

features, which will be detailed in section 4.1. With twelve features and around 75 categorical feature values, the state space is still huge. The second state-space reduction is to use a decision tree to learn a mapping from state representations to actions. Each leaf of the tree is an equivalence class over feature vectors and therefore also over complete states. The action assigned to the leaf is the highest-frequency action found in the training instances that map to the leaf. Thus we have three different notions of state: the complete state, the state representation, and the state equivalence classes at the leaves of the decision tree.

We use a CART-style decision tree algorithm (Brieman, et al., 1984) as our main machine learning tool. The training sets used here contained over two million instances. The CART algorithm was modified to handle large training sets by using samples, rather than all instances, to choose tests at each node of the tree. All features used were categorical, and all tests were binary. Our decision trees had roughly 20k leaves. Tree induction took around twenty minutes.

3.3 Markov Parsing Models

We define a class of conditional distributions over parse trees which we call Markov parsing models (MPMs). Consider a reversible parsing automaton which takes a sequence of n actions $(a_1, a_2, ...a_n)$ on an input string σ to produce a parse t. At each step, the automaton is in some state s_i, and in every state the automaton chooses actions according to a stochastic policy $P(a_i|s_i)$. Because of reversibility, for a given input σ, there is an isomorphism between parse trees and action sequences:

$$t \Leftrightarrow (a_1, a_2, ...a_n)$$

Taking probabilities,

$$\begin{aligned} P(t|\sigma) &= P(a_1, a_2, ...a_n|\sigma) && (1) \\ &= \prod_{i=1}^{n} P(a_i|a_{i-1}...a_1, \sigma) && (2) \\ &= \prod_{i=1}^{n} P(a_i|s_i) && (3) \end{aligned}$$

The second step merely rewrites equation 1 using a probailistic identity. In the third step, replacing the history at the ith time step $(a_{i-1}, ...a_1, \sigma)$ with the state s_i is an expression of the Markov property. This is justified since for a reversible automaton, the action sequence defines a unique state, and that state could only be reached by that action sequence.

Equation 3 defines a Markov parsing model. Generative models, such as PCFGs, define a joint distribution $P(t, \sigma)$ over trees and strings. By contrast, a parsing model defines $P(t|\sigma)$, conditioned on the input string. Assuming that the input string is given, a potential advantage of a conditional model over a generative one is that it makes better use of limited training data, as it doesn't need to model the string probability. The string probability is useful in some applications, such as speech recognition, but it requires extra parameters and training data to model it.

An MPM plays two roles. First, as in most statistical parsers, it facilitates syntactic disambiguation by specifying the relative likelihood of the various structures which might be assigned to a sentence. Second, it is directly useful for control; it tells us how to parse and search efficiently. By contrast, in some recent models (Collins, 1999; Charniak et al. 1998), some events used in the model are not available until after decisions are made; therefore a separate "figure of merit" must be engineered to guide search.

3.4 ML Estimation of MPM parameters

The parameters of an MPM can be estimated using a treebank. Consider the decision-tree induction procedure described in section 3.2. Each leaf of the tree corresponds to a state s in the model, and contains a set of training instances. For each action a, the ML estimate of $P(a|s)$ is simply the relative frequency of that action in the training instances at that leaf.

A similar distribution can be defined for any node in the tree, not just for leaves. If necessary, the ML estimates can be smoothed by "backing off" to the distribution at the next-higher level in the tree. Other smoothing methods are possible as well.

4 Description of parsers

We now describe the parsers we implemented. Three parsing strategies (top-down, left-corner, and shift-reduce) have been discussed extensively in the literature. As there is no consensus on which is best for parsing natural language, we tried all three. Our goal was not to directly compare the strategies, but simply to find the

one that worked best in our system. Direct comparison would be difficult, in particular because the choice of state representation has a big influence on performance; and there is no obvious way of choosing the best state representation for a particular parsing strategy.

The input sentences were pre-tagged using the MAXPOST tagger (Ratnaparkhi, 1996). All parsers here are unlexicalized, so they use preterminals (part-of-speech tags) as their input symbols. Each parser has an input (or lookahead) buffer, organized as a FIFO queue. Each parser also has a stack. Stack items are labeled with either a preterminal symbol or a nonterminal (a syntactic category). The "completeness" of a stack item is different in the three parsing strategies (a node is considered complete if it is connected to its yield). Below, for conciseness, we describe some actions as having arguments; this is shorthand for the set of actions containing each distinct combination of arguments. All three parsers handle failure as described in section 3.1, that is, by returning a FAIL node whose children are the constituents completed so far, plus any remaining input.

Even within a parsing strategy, we have considerable latitude in designing the dynamics of a parser. For example, Roark (2001) describes how a top-down parser can be aggressive or lazy. It is advantageous to be lazy, since delayed predictions are made when there is better evidence for the correct prediction. For this and other reasons, the parsers described below depart somewhat from the usual textbook definitions.

Shift-Reduce: The SR parser's SHIFT action dequeues an input item and pushes it on the stack. The REDUCE(n, CAT) action pops n stack symbols ($n \geq 1$), makes them children of a new symbol labeled CAT, and pushes that symbol on the stack. The SR parser terminates when the input is consumed and the stack contains the special symbol TOP. In the SR parser, all stack items are always complete; the tree under a stack node is not modified further.

Top-Down: The TD parser has a PREDICT(*list*) action, where the elements of *list* are either terminals or nonterminals. The PREDICT action pops the stack, makes a new item for each list element, pushes each of these on the stack in reverse order, and makes each new item a child of the popped item. The other action is MATCH. This action is performed if and only if the top-of-stack item is a preterminal. The stack is popped, one input symbol is dequeued, and the popped stack item is replaced in the tree by the input item. (Our MATCH coerces the prediction to use the input label, rather than requiring a match, which causes too many failures.) In the TD parser, all stack items are predictions, and are incomplete in the sense that their yield has not been matched yet.

Left-Corner: Unlike the other two strategies, the LC parser's stack may contain both complete and incomplete items. Every incomplete item is marked as such. Also, every incomplete item has a complete left-corner (that is, left-most child). The LC parser has three actions. SHIFT is the same as for SR. The PROJECT(CAT) action pops a completed item from the stack, and makes it the left corner of a new incomplete node labeled CAT, which is pushed onto the stack. Finally, the ATTACH action finds the first incomplete item on the stack, pops all items above it, makes them its children, and marks the stack node, which is now at the top of the stack, as complete.

4.1 Representation

Treebank Representation: Following many previous researchers, we binarize the treebank, as illustrated in Fig. 4.1. There are several reasons for doing this. The Penn Treebank employs a very flat tree style, particularly for noun phrases. Some nodes have eight or more children. For the SR and LC parsers, this means that many words must be shifted onto the stack before a REDUCE or ATTACH action. Binarization breaks a single decision into a sequence of smaller ones. Also, the parser's data structures are used in a more uniform way, allowing for improvements in state representation. For example, in binarized SR parsing, the top two stack nodes are the only candidates for reduction, and the previous stack node always represents the phrase preceding the one being built. For TD, binarization has the effect of delaying predictions. Roark (2001) showed that this is a big advantage for top-down parsing, particularly right binarization to nullary. We tried several binarization transformations. Unlike previous work, we labeled all nodes introduced by binarization as e.g. NP*, simply noting that this is a "synthetic" child of an NP. These binarizations are reversible, and we convert back to Penn Treebank style before evaluation.

State representation: The state representation for each parser consisted of twelve cate-

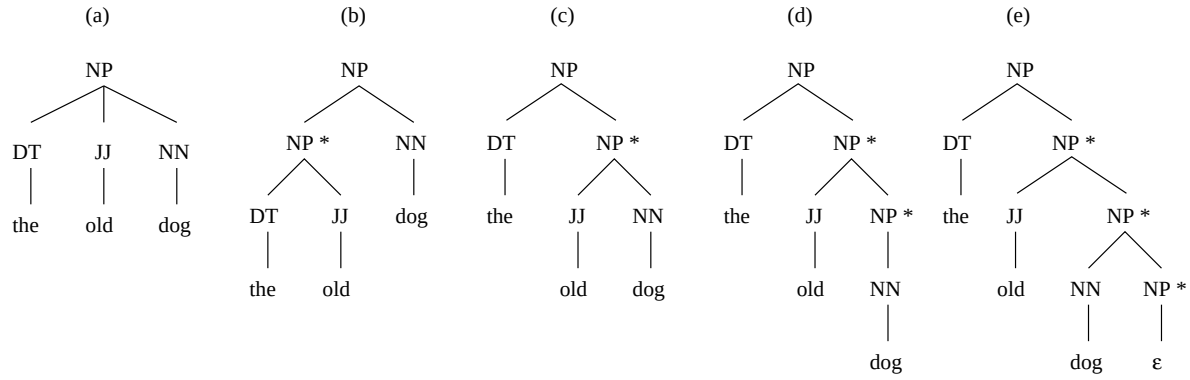

Figure 1: Tree binarizations: (a) original; (b) left binarized (L); (c) right binarized to binary (R2); (d) right binarized to unary (R1); (e) right binarized to nullary (R0)

gorical features. Each feature is a node label, either a non-terminal (POS tag) or a terminal symbol (syntactic category). There are 49 distinct POS tags and 28 distinct category labels. We attempted to choose the items that would be the most relevant to the parsing decisions. The choices represented here are based on intuition along with trial and error; no systematic attempt has been made so far to determine the best set of features for the state representations. This is an area for future work. We used the same number (twelve) for each of the three parsers to make them roughly comparable.

Each parser's state representation contained features for the first four input symbols and the top two stack items. The remaining features are as follows:

SR: the third and fourth stack items, and the left and right children of the first two stack items.

LC: the third and fourth stack items, the left and right children of the first stack item, and the left children of the second and third stack items.

TD: the first four ancestors of the first stack item, and the first two completed phrases preceding the first stack item (found by going to the parent, then to its left child, returning it if the child is complete, otherwise recursing on the parent).

The choice of items to include in the state representation corresponds to choosing events for the probabilistic models used in other statistical parsers. The different parsing strategies provide different opportunities for conditioning on context. This is a very rich topic which unfortunately we can't explore further here.

5 Results

All experiments were done on the standard Penn Treebank Wall St. Journal task (Marcus et al., 1993), for comparison with other work. We used sections 2-21 for training, section 0 for development testing, and section 23 for final testing. All preliminary experiments used the development set for testing. We evaluated performance of each parser with several treebank transforms. Results are in Table 1. We report recall and precision for all sentences with length ≤ 100 and for all sentences with length ≤ 40 tokens. For a treebank parse T and a parse t to be evaluated, these measures are defined as

$$\text{recall} = \frac{\#\text{ correct constituents in } t}{\#\text{ constituents in } T}$$

$$\text{precision} = \frac{\#\text{ correct constituents in } t}{\#\text{ constituents in } t}$$

We followed the standard practice of ignoring punctuation and conflating ADVP and PRN for purposes of evaluation. The results reported are for all results, not just complete parses. For FAIL nodes, the evaluation measures give partial credit for whatever has been completed correctly. Including incomplete parses in the results tends to lower recall and precision, compared to the results for the complete parses only.

Coverage: Coverage is the fraction of the test set for which the parser found a complete parse. The parsers here always return a parse tree, but some of those trees represent parse failure, as noted earlier. The SR-L and LC-R2 parsers have almost complete coverage, with

Parser	Transform	Coverage	length ≤ 100		length ≤ 40		Words per second
			Recall	Precision	Recall	Precision	
SR	L	99.8	76.7	75.8	77.8	77.0	33,740
SR	R2	94.9	75.9	77.2	77.1	78.2	33,560
SR	R1	90.8	75.6	77.3	76.9	78.3	28,398
LC	L	95.6	71.9	71.9	72.9	72.8	25,812
LC	R2	99.9	73.9	74.0	74.9	75.0	24,948
LC	R1	96.2	74.4	74.3	75.6	75.4	21,610
TD	L	31.0	38.7	57.1	41.3	58.3	41,740
TD	R2	42.3	47.6	61.6	50.2	62.6	45,274
TD	R1	72.0	61.5	66.8	62.9	68.2	30,739
TD	R0	98.4	69.3	72.1	70.6	73.2	21,341

Table 1: Parser performance on section 23 of the Penn Treebank. Coverage, recall, and precision are given as percentages.

TD-R0 lagging slightly behind. As in Roark (2001), increasingly aggressive binarization is beneficial for top-down parsing, because decisions are delayed. For greedy parsers with coverage in the high nineties, complete coverage could be attained at minimal additional cost by using search only for sentences where the greedy parse produced a parse failure.

Accuracy: The best recall and precision reported here are better than a basic treebank PCFG, for which Johnson (1998) gives 69.7% and 73.5% respectively (for length ≤ 100), under identical conditions. Our results are considerably below the state of the art for this task, currently around 90%, which is achieved with much more sophisticated probabilistic models. Considering their speed and the simplicity of their representations, it is remarkable that our parsers achieve the levels of accuracy and coverage reported here. Even at these speeds, improvements in accuracy may be possible by improving the representation. And of course, accuracy could be improved at the expense of speed by adding search (see section 6). The TD parser lags substantially behind SR in accuracy. The accuracy problem for TD and its slightly worse coverage are probably due to the same cause. We suspect that predictive parsing is inherently riskier than bottom-up parsing. Unlike the other two strategies, predictions must sometimes be made when there is no immediately adjacent complete node in the tree. However, these comparisons are not conclusive, because the choice of features for the state representation may also have an important role in the differences.

Speed: Parsing speeds are reported in words per second. This is exclusive of tagging time (recall that we pre-tagged our input), and also exclusive of IO. Experiments were done on a 1.2 GHz Athlon CPU with 1.25 GB of memory running Linux. The parsers were implemented in Java, including the decision tree module. The JVM version was 1.4.2, and the JVM was warmed up before testing for speed. No additional effort was spent on speed optimization. Clearly, these speeds are quite fast. A fast contemporary tagger, TnT (Brants, 2000), which is implemented in C, tags between 30,000 and 60,000 words per second running on a Pentium 500 MHz CPU.

Our LC parser is slightly slower than our SR and TD parsers because LC inherently makes more decisions per sentence than the others do. Speeds for the low-accuracy TD runs are high due to the fact that the parser stops early when it encounters a failure. Comparing these speeds with other statistical parsers is somewhat problematic. Differences in CPU speeds and implementation languages obscure the comparison. Moreover, many authors simply report accuracy measures, and don't report timing results. Any deterministic parser will have running time that is linear in the size of the input, and the amount of work per input word that needs to be done is small, dominated by the decision tree module, which is not expensive. By contrast, most current statistical parsers lean towards the other end of the speed-accuracy tradeoff spectrum.

One paper that focuses on efficiency of statistical parsing is Charniak et al. (1998). They used a chart parser, and measured speed in units of popped edges per sentence. This corresponds closely to the number of actions per sentence

taken by a parsing automaton. They report that on average the minimum number of popped edges to create a correct parse would be 47.5. By this measure, our greedy parsers would take on average very close to 47 actions. They report 95% coverage and 75% average recall and precision on sentences of length ≤ 40 with 490 popped edges; this is ten times the minimum number of steps. However, to get complete coverage, they required 1760 popped edges, which is a factor of 37 greater than the minimum.

Wong and Wu (1999) report recall and precision of 78.9% and 77.7% respectively for their deterministic shift-reduce parser on sentences of length ≤ 40, which is very similar to the accuracy of our SR-L run. They reported a rate of 528 words per second, but did not specify the hardware configuration.

6 Future work

The approach described here can be extended in a number of ways. As noted, a Markov parsing model can be used to guide search. We plan to add a beam search to explore the speed-accuracy tradeoff. Improvements in the state representation are possible, particularly along the lines of linguistically-motivated treebank transformations, as in Klein and Manning (2003). Adding a lexical component to the model is another extension we intend to investigate.

7 Conclusions

Deterministic unlexicalized statistical parsers have surprisingly good accuracy and coverage, considering their speed and simplicity. The best parsers reported here have almost complete coverage, outperform basic PCFGs, and are roughly as fast as taggers. We described an approach to statistical parsing based on induction of stochastic automata. We defined Markov parsing models, described how to estimate parameters for them, and showed how the deterministic parsers we implemented are greedy versions of MPM parsers. We found that for greedy parsing, bottom-up parsing strategies seem to have a small advantage over top-down.

Acknowledgements

Thanks to Brian Roark for helpful comments on this paper.

References

Steven Abney, David McAllester, and Fernando Pereira. 1999. Relating Probabilistic Grammars and Automata. *37th Annual Meeting of the Association for Computational Linguistics: Proceedings of the Conference*, pp. 542-549.

E. Black, F. Jelinek, J. Lafferty, D. M. Magerman, R. Mercer, and S. Roukos. 1992. Towards History-based Grammars: Using Richer Models for Probabilistic Parsing. *Proceedings of the DARPA Speech and Natural Language Workshop.*

Thorsten Brants. 2000. TnT – A Statistical Part-of-Speech Tagger. *Proceedings of the Sixth Applied Natural Language Processing Conference.*

Leo Brieman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. 1984. *Classification and Regression Trees.* Chapman & Hall.

Eugene Charniak. 2000. A Maximum-Entropy-Inspired Parser. In *Proceedings of the 1st Conference of the North American Chapter of the Association for Computational Linguistics.*

Eugene Charniak, Sharon Goldwater, and Mark Johnson. 1998. Edge-based best-first chart parsing. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 127–133

Michael Collins. 1999. Head-Driven Statistical Models for Natural Language Parsing. Ph.D. Dissertation, University of Pennsylvania.

Jesus Gimenez and Luis Marquez. 2003. Fast and Accurate Part-of-Speech Tagging: The SVM Approach Revisited . *Recent Advances in Natural Language Processing.*

Ulf Hermjakob. 1997. Learning Parse and Translation Decisions from Examples with Rich Context. Ph.D. Dissertation, University of Texas.

Mark Johnson. 1998. PCFG models of linguistic tree representations. *Computational Linguistics*, 24(4):617-636.

Dan Klein and Christopher D. Manning. 2002. Fast Exact Natural Language Parsing with a Factored Model. *Advances in Neural Information Processing Systems 15.*

Dan Klein and Christopher D. Manning. 2003. Accurate Unlexicalized Parsing. *41st Annual Meeting of the Association for Computational Linguistics: Proceedings of the Conference.*

David M. Magerman. 1994. Natural Language Parsing as Statistical Pattern Recognition. Ph.D. Dissertation, Stanford University.

Mitchell P. Marcus, Beatrice Santorini, and

Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19:313–330.

Adwait Ratnaparkhi. 1996. A Maximum Entropy Part-Of-Speech Tagger. In *Proceedings of the 1st Conference on Empirical Methods in Natural Language Processing*.

Adwait Ratnaparkhi. 1998. Maximum Entropy Models for Natural Language Ambiguity Resolution. Ph.D. Dissertation. University of Pennsylvania.

Brian Roark. 2001. Robust Probabilistic Predictive Syntactic Processing: Motivations, Models, and Applications. Ph.D. dissertation. Brown University.

Aboy Wong and Dekai Wu. 1999. Learning a lightweight robust deterministic parser. *Sixth European Conference on Speech Communication and Technology*.