Computational Challenges in Parsing by Classification

Joseph Turian and I. Dan Melamed
{lastname}@cs.nyu.edu
Computer Science Department
New York University
New York, New York 10003

Abstract

This paper presents a discriminative parser that does not use a generative model in any way, yet whose accuracy still surpasses a generative baseline. The parser performs feature selection incrementally during training, as opposed to *a priori*, which enables it to work well with minimal linguistic cleverness. The main challenge in building this parser was fitting the training data into memory. We introduce gradient sampling, which increased training speed 100-fold. Our implementation is freely available at http://nlp.cs.nyu.edu/parser/.

1 Introduction

Discriminative machine learning methods have improved accuracy on many NLP tasks, including POS-tagging, shallow parsing, relation extraction, and machine translation. However, only limited advances have been made on full syntactic constituent parsing. Successful discriminative parsers have used generative models to reduce training time and raise accuracy above generative baselines (Collins & Roark, 2004; Henderson, 2004; Taskar et al., 2004). However, relying upon information from a generative model might limit the potential of these approaches to realize the accuracy gains achieved by discriminative methods on other NLP tasks. Another difficulty is that discriminative parsing approaches can be very task-specific and require quite a bit of trial and error with different hyper-parameter values and types of features.

In the present work, we make progress towards overcoming these obstacles. We propose a flexible, well-integrated method for training discriminative parsers, demonstrating techniques that might also be useful for other structured learning problems. The learning algorithm projects the hand-provided atomic features into a compound feature space and performs incremental feature selection from this large feature space. We achieve higher accuracy than a generative baseline, despite not using the standard trick of including an underlying generative model. Our training regime does model selection without ad-hoc smoothing or frequency-based feature cutoffs, and requires no heuristics to optimize the single hyper-parameter.

We discuss the computational challenges we overcame to build this parser. The main difficulty is that the training data fit in memory only using an indirect representation,[1] so the most costly operation during training is accessing the features of a particular example. We show how to train a parser effectively under these conditions. We also show how to speed up training by using a principled sampling method to estimate the loss gradients used in feature selection.

§2 describes the parsing algorithm. §3 presents the learning method and techniques used to reduce training time. §4 presents experiments with discriminative parsers built using these methods. §5 dis-

[1] Similar memory limitations exist in other large-scale NLP tasks. Syntax-driven SMT systems are typically trained on an order of magnitude more sentences than English parsers, and unsupervised estimation methods can generate an arbitrary number of negative examples (Smith & Eisner, 2005).

Workshop on Computationally Hard Problemsand Joint Inference in Speech and Language Processing, pages 17–24,
New York City, New York, June 2006.

cusses possible issues in scaling to larger example sets.

2 Parsing Algorithm

The following terms will help to explain our work. A *span* is a range over contiguous words in the input. Spans *cross* if they overlap but neither contains the other. An *item* is a (span, label) pair. A *state* is a partial parse, i.e. a set of items, none of whose spans cross. A parse *inference* is a (state, item) pair, i.e. a state and a (consequent) item to be added to it. The *frontier* of a state consists of the items with no parents yet. The *children* of an inference are the frontier items below the item to be inferred, and the *head* of an inference is the child item chosen by head rules (Collins, 1999, pp. 238–240). A parse *path* is a sequence of parse inferences. For some input sentence and training parse tree, a state is *correct* if the parser can infer zero or more additional items to obtain the training parse tree and an inference is correct if it leads to a correct state.

Now, given input sentence s we compute:

$$\hat{p} = \arg\min_{p \in P(s)} \left(\sum_{i \in p} l(i) \right) \tag{1}$$

where $P(s)$ are possible parses of the sentence, and the *loss* (or cost) l of parse p is summed over the inferences i that lead to the parse. To find $\hat{p}$, the parsing algorithm considers a sequence of states. The initial state contains terminal items, whose labels are the POS tags given by Ratnaparkhi (1996). The parser considers a set of (bottom-up) inferences at each state. Each inference results in a successor state to be placed on the agenda. The loss function l can consider arbitrary properties of the input and parse state,[2] which precludes a tractable dynamic programming solution to Equation 1. Therefore, we do standard agenda-based parsing, but instead of items our agenda stores entire states, as per more general best-first search over parsing hypergraphs (Klein & Manning, 2001). Each time we pop a state from the agenda, l computes a loss for the bottom-up inferences generated from that state. If the loss of the popped state exceeds that of the current best complete parse, search is done and we have found the optimal parse.

[2] I.e. we make no context-free assumptions.

3 Training Method

3.1 General Setting

From each training inference $i \in I$ we generate the tuple $\langle X(i), y(i), b(i) \rangle$. $X(i)$ is a feature vector describing i, with each element in $\{0, 1\}$. The observed y-value $y(i) \in \{-1, +1\}$ is determined by whether i is a correct inference or not. Some training examples might be more important than others, so each is given an initial bias $b(i) \in \mathbb{R}^+$.

Our goal during training is to induce a real-valued inference scoring function (hypothesis) $h(i; \boldsymbol{\alpha})$, which is a linear model parameterized by a vector $\boldsymbol{\alpha}$ of reals:

$$h(i; \boldsymbol{\alpha}) = \boldsymbol{\alpha} \cdot X(i) = \sum_f \alpha_f \cdot X_f(i) \tag{2}$$

Each f is a feature. The sign of $h(i; \boldsymbol{\alpha})$ predicts the y-value of i and the magnitude gives the confidence in this prediction.

The training procedure optimizes $\boldsymbol{\alpha}$ to minimize the expected risk R:

$$R(I; \boldsymbol{\alpha}) = L(I; \boldsymbol{\alpha}) + \Omega(\boldsymbol{\alpha}) \tag{3}$$

In principle, L can be any loss function, but in the present work we use the log-loss (Collins et al., 2002):

$$L(I; \boldsymbol{\alpha}) = \sum_{i \in I} l(i; \boldsymbol{\alpha}) = \sum_{i \in I} b(i) \cdot \sigma(\mu(i; \boldsymbol{\alpha})) \tag{4}$$

where:

$$\sigma(\mu) = \ln(1 + \exp(-\mu)) \tag{5}$$

and the *margin* of inference i under the current model $\boldsymbol{\alpha}$ is:

$$\mu(i; \boldsymbol{\alpha}) = y(i) \cdot h(i; \boldsymbol{\alpha}) \tag{6}$$

For a particular choice of $\boldsymbol{\alpha}$, $l(i)$ in Equation 1 is computed according to Equation 4 using $y(i) = +1$ and $b(i) = 1$.

$\Omega(\boldsymbol{\alpha})$ in Equation 3 is a regularizer, which penalizes overly complex models to reduce overfitting and generalization error. We use the ℓ_1 penalty:

$$\Omega(\boldsymbol{\alpha}) = \sum_f \lambda \cdot |\alpha_f| \tag{7}$$

where λ is the ℓ_1 parameter that controls the strength of the regularizer. This choice of objective R is motivated by Ng (2004), who suggests that, given a

learning setting where the number of irrelevant features is exponential in the number of training examples, we can nonetheless learn effectively by building decision trees to minimize the ℓ_1-regularized log-loss. Conversely, Ng (2004) suggests that most of the learning algorithms commonly used by discriminative parsers *will* overfit when exponentially many irrelevant features are present.[3]

Learning over an exponential feature space is the very setting we have in mind. *A priori*, we define only a set A of simple *atomic* features (see §4). However, the learner induces *compound* features, each of which is a conjunction of possibly negated atomic features. Each atomic feature can have three values (yes/no/don't care), so the size of the compound feature space is $3^{|A|}$, exponential in the number of atomic features. It was also exponential in the number of training examples in our experiments ($|A| \approx |I|$).

We use an ensemble of confidence-rated decision trees (Schapire & Singer, 1999) to represent h.[4] Each node in a decision tree corresponds to a compound feature, and the leaves of the decision trees keep track of the parameter values of the compound features they represent. To score an inference using a decision tree, we percolate the inference down to a leaf and return that leaf's confidence. The overall score given to an inference by the whole ensemble is the sum of the confidences returned by the trees in the ensemble.

3.2 Boosting ℓ_1-Regularized Decision Trees

Listing 1 presents our training algorithm. (Sampling will be explained in §3.3. Until then, assume that the sample S is the entire training set I.) At the beginning of training, the ensemble is empty, $\boldsymbol{\alpha} = \mathbf{0}$, and the ℓ_1 parameter λ is set to ∞. We train until the objective cannot be further reduced for the current choice of λ. We then relax the regularization penalty by decreasing λ and continuing training. We also determine the accuracy of the parser on a held-out development set using the previous λ value (before it was decreased), and can stop training when this accuracy plateaus. In this way, instead of choosing the best λ heuristically, we can optimize it during a single training run (Turian & Melamed, 2005).

Listing 1 Training algorithm.

```
1:  procedure TRAIN(I)
2:      ensemble ← ∅
3:      h(i) ← 0 for all i ∈ I
4:      for T = 1 ... ∞ do
5:          S ← priority sample I
6:          extract X(i) for all i ∈ S
7:          build decision tree t using S
8:          percolate every i ∈ I to a leaf node in t
9:          for each leaf f in t do
10:             choose α_f to minimize R
11:             add α_f to h(i) for all i in this leaf
```

Our strategy for optimizing $\boldsymbol{\alpha}$ to minimize the objective R (Equation 3) is a variant of steepest descent (Perkins et al., 2003). Each training iteration has several steps. First, we choose some new compound features that have high magnitude gradient with respect to the objective function. We do this by building a new decision tree, whose leaves represent the new compound features.[5] Second, we confidence-rate each leaf to minimize the objective over the examples that percolate down to that leaf. Finally, we append the decision tree to the ensemble and update parameter vector $\boldsymbol{\alpha}$ accordingly. In this manner, compound feature selection is performed incrementally *during* training, as opposed to *a priori*.

To build each decision tree, we begin with a root node, and we recursively split nodes by choosing a splitting feature that will allow us to decrease the objective. We have:

$$\frac{\partial L(I;\boldsymbol{\alpha})}{\partial \alpha_f} = \sum_{i \in I} \frac{\partial l(i;\boldsymbol{\alpha})}{\partial \mu(i;\boldsymbol{\alpha})} \cdot \frac{\partial \mu(i;\boldsymbol{\alpha})}{\partial \alpha_f} \tag{8}$$

where:

$$\frac{\partial \mu(i;\boldsymbol{\alpha})}{\partial \alpha_f} = y(i) \cdot X_f(i) \tag{9}$$

We define the *weight* of an example under the current model as:

$$w(i;\boldsymbol{\alpha}) = -\frac{\partial l(i;\boldsymbol{\alpha})}{\partial \mu(i;\boldsymbol{\alpha})} = b(i) \cdot \frac{1}{1 + \exp(\mu(i;\boldsymbol{\alpha}))}. \tag{10}$$

[3] including the following learning algorithms:
- unregularized logistic regression
- logistic regression with an ℓ_2 penalty (i.e. a Gaussian prior)
- SVMs using most kernels
- multilayer neural nets trained by backpropagation
- the perceptron algorithm

[4] Turian and Melamed (2005) show that that decision trees applied to parsing have higher accuracy and training speed than decision stumps.

[5] Any given compound feature can appear in more than one tree.

and:

$$W_f^{\bar{y}}(I;\boldsymbol{\alpha}) = \sum_{\substack{i \in I \\ X_f(i)=1, y(i)=\bar{y}}} w(i;\boldsymbol{\alpha}) \tag{11}$$

Combining Equations 8–11 gives:[6]

$$\frac{\partial L}{\partial \alpha_f} = W_f^{-1} - W_f^{+1} \tag{12}$$

We define the *gain* G_f of feature f as:

$$G_f = \max\left(0, \left|\frac{\partial L}{\partial \alpha_f}\right| - \lambda\right) \tag{13}$$

Equation 13 has this form because the gradient of the penalty term is undefined at $\alpha_f = 0$. This discontinuity is why ℓ_1 regularization tends to produce sparse models. If $G_f = 0$, then the objective R is at its minimum with respect to parameter α_f. Otherwise, G_f is the magnitude of the gradient of the objective as we adjust α_f in the appropriate direction.

The gain of splitting node f using some atomic feature a is defined as

$$\check{G}_f(a) = G_{f \wedge a} + G_{f \wedge \neg a} \tag{14}$$

We allow node f to be split only by atomic features a that increase the gain, i.e. $\check{G}_f(a) > G_f$. If no such feature exists, then f becomes a leaf node of the decision tree and α_f becomes one of the values to be optimized during the parameter update step. Otherwise, we choose atomic feature $\hat{a}$ to split node f:

$$\hat{a} = \arg\max_{a \in A} \check{G}_f(a) \tag{15}$$

This split creates child nodes $f \wedge \hat{a}$ and $f \wedge \neg\hat{a}$. If no root node split has positive gain, then training has converged for the current choice of ℓ_1 parameter λ.

Parameter update is done sequentially on only the most recently added compound features, which correspond to the leaves of the new decision tree. After the entire tree is built, we percolate examples down to their appropriate leaf nodes. We then choose for each leaf node f the parameter α_f that minimizes the objective R over the examples in that leaf. Decision trees ensure that these compound features are mutually exclusive, so they can be directly optimized independently of each other using a line search over the objective R.

[6] Since $\boldsymbol{\alpha}$ is fixed during a particular training iteration and I is fixed throughout training, we omit parameters $(I;\boldsymbol{\alpha})$ henceforth.

3.3 Sampling for Faster Feature Selection

Building a decision tree using the entire example set I can be very expensive, which we will demonstrate in §4.2. However, feature selection can be effective even if we don't examine every example. Since the weight of high-margin examples can be several orders of magnitude lower than that of low-margin examples (Equation 10), the contribution of the high-margin examples to feature weights (Equation 11) will be insignificant. Therefore, we can ignore most examples during feature selection as long as we have good estimates of feature weights, which in turn give good estimates of the loss gradients (Equation 12).

As shown in Step 1.5 of Listing 1, before building each decision tree we use priority sampling (Duffield et al., 2005) to choose a small subset of the examples according to the example weights given by the current classifier, and the tree is built using only this subset. We make the sample small enough that its entire atomic feature matrix will fit in memory. To optimize decision tree building, we compute and cache the sample's atomic feature matrix in advance (Step 1.6).

Even if the sample is missing important information in one iteration, the training procedure is capable of recovering it from samples used in subsequent iterations. Moreover, even if a sample's gain estimates are inaccurate and the feature selection step chooses irrelevant compound features, confidence updates are based upon the entire training set and the regularization penalty will prevent irrelevant features from having their parameters move away from zero.

3.4 The Training Set

Our training set I contains all inferences considered in every state along the correct path for each gold-standard parse tree (Sagae & Lavie, 2005).[7] This method of generating training examples does not require a working parser and can be run prior to any training. The downside of this approach is that it minimizes the error of the parser at *correct* states only. It does not account for compounded error or teach the parser to recover from mistakes gracefully.

[7] Since parsing is done deterministically right-to-left, there can be no more than one correct inference at each state.

Turian and Melamed (2005) observed that uniform example biases $b(i)$ produced lower accuracy as training progressed, because the induced classifiers minimized the *example-wise* error. Since we aim to minimize the state-wise error, we express this bias by assigning every training *state* equal value, and—for the examples generated from that state—sharing half the value uniformly among the negative examples and the other half uniformly among the positive examples.

Although there are $O(n^2)$ possible spans over a frontier containing n items, we reduce this to the $O(n)$ inferences that cannot have more than 5 children. With no restriction on the number of children, there would be $O(n^2)$ bottom-up inferences at each state. However, only 0.57% of non-terminals in the preprocessed development set have more than five children.

Like Turian and Melamed (2005), we parallelize training by inducing 26 label classifiers (one for each non-terminal label in the Penn Treebank). Parallelization might not uniformly reduce training time because different label classifiers train at different rates. However, parallelization uniformly reduces *memory* usage because each label classifier trains only on inferences whose consequent item has that label. Even after parallelization, the atomic feature matrix cannot be cached in memory. We can store the training inferences in memory using only an *indirect* representation. More specifically, for each inference i in the training set, we cache in memory several values: a pointer i to a tree cut, its y-value $y(i)$, its bias $b(i)$, and its confidence $h(i)$ under the current model. We cache $h(i)$ throughout training because it is needed both in computing the gradient of the objective during decision tree building (Step 1.7) as well as subsequent minimization of the objective over the decision tree leaves (Step 1.10). We update the confidences at the end of each training iteration using the newly added tree (Step 1.11).

The most costly operation during training is to access the feature values in $X(i)$. An atomic feature *test* determines the value $X_a(i)$ for a single atomic feature a by examining the tree cut pointed to by inference i. Alternately, we can perform atomic feature *extraction*, i.e. determine *all* non-zero atomic features over i.[8] Extraction is 100–1000 times more expensive than a single test, but is necessary during decision tree building (Step 1.7) because we need the entire vector $X(i)$ to accumulate inferences in children nodes. Essentially, for each inference i that falls in some node f, we accumulate $w(i)$ in $W^{y(i)}_{f \wedge a}$ for all a with $X_a(i) = 1$. After all the inferences in a node have been accumulated, we try to split the node (Equation 15). The negative child weights are each determined as $W^{y}_{f \wedge \neg a} = W^{y}_{f} - W^{y}_{f \wedge a}$.

4 Experiments

We follow Taskar et al. (2004) and Turian and Melamed (2005) in training and testing on ≤ 15 word sentences in the English Penn Treebank (Taylor et al., 2003). We used sections 02–21 for training, section 22 for development, and section 23, for testing. We use the same preprocessing steps as Turian and Melamed (2005): during both training and testing, the parser is given text POS-tagged by the tagger of Ratnaparkhi (1996), with capitalization stripped and outermost punctuation removed.

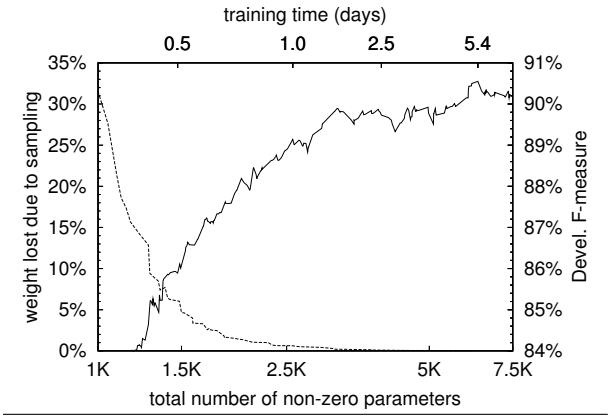
For reasons given in Turian and Melamed (2006), items are inferred bottom-up right-to-left. As mentioned in §2, the parser cannot infer any item that crosses an item already in the state. To ensure the parser does not enter an infinite loop, no two items in a state can have both the same span and the same label. Given these restrictions, there were roughly 40 million training examples. These were partitioned among the constituent label classifiers.

Our atomic feature set A contains features of the form "is there an item in group J whose label/headword/headtag/headtagclass[9] is 'X'?". Possible values of 'X' for each predicate are collected from the training data. Some examples of possible values for J include the last n child items, the first n left context items, all right context items, and the terminal items dominated by the non-head child items. Space constraints prevent enumeration of the headtagclasses and atomic feature templates, which are

[8] Extraction need not take the naïve approach of performing $|A|$ different tests, and can be optimized by using knowledge about the nature of the atomic feature templates.

[9] The predicate headtagclass is a supertype of the headtag. Given our compound features, these are not strictly necessary, but they accelerate training. An example is "proper noun," which contains the POS tags given to singular and plural proper nouns.

Figure 1 F_1 score of our parser on the development set of the Penn Treebank, using only ≤ 15 word sentences. The dashed line indicates the percent of NP example weight lost due to sampling. The bottom *x*-axis shows the number of non-zero parameters in each parser, summed over all label classifiers.



instead provided at the URL given in the abstract. These templates gave 1.1 million different atomic features. We experimented with smaller feature sets, but found that accuracy was lower. Charniak and Johnson (2005) use linguistically more sophisticated features, and Bod (2003) and Kudo et al. (2005) use sub-tree features, all of which we plan to try in future work.

We evaluated our parser using the standard PARSEVAL measures (Black et al., 1991): labelled precision, labelled recall, and labelled F-measure (Prec., Rec., and F_1, respectively), which are based on the number of non-terminal items in the parser's output that match those in the gold-standard parse. The solid curve Figure 1 shows the accuracy of the parser over the development set as training progressed. The parser exceeded 89% F-measure after 2.5 days of training. The peak F-measure was 90.55%, achieved at 5.4 days using 6.3K active parameters. We omit details given by Turian and Melamed (2006) in favor of a longer discussion in §4.2.

4.1 Test Set Results

To situate our results in the literature, we compare our results to those reported by Taskar et al. (2004) and Turian and Melamed (2005) for their discriminative parsers, which were also trained and tested on ≤ 15 word sentences. We also compare our parser to a representative non-discriminative parser (Bikel, 2004),[10] the only one that we were able to train and test under exactly the same experimental conditions (including the use of POS tags from Ratnaparkhi (1996)). Table 1 shows the PARSEVAL results of these four parsers on the test set.

Table 1 PARSEVAL results of parsers on the test set, using only ≤ 15 word sentences.

	F_1 %	Rec. %	Prec. %
Turian and Melamed (2005)	87.13	86.47	87.80
Bikel (2004)	88.30	87.85	88.75
Taskar et al. (2004)	89.12	89.10	89.14
our parser	89.40	89.26	89.55

Table 2 Profile of an NP training iteration, given in seconds, using an AMD Opteron 242 (64-bit, 1.6Ghz). Steps refer to Listing 1.

Step	Description	mean	stddev	%
1.5	Sample	1.5s	0.07s	0.7%
1.6	Extraction	38.2s	0.13s	18.6%
1.7	Build tree	127.6s	27.60s	62.3%
1.8	Percolation	31.4s	4.91s	15.3%
1.9–11	Leaf updates	6.2s	1.75s	3.0%
1.5–11	Total	204.9s	32.6s	100.0%

4.2 Efficiency

40% of non-terminals in the Penn Treebank are NPs. Consequently, the bottleneck in training is induction of the NP classifier. It was trained on 1.65 million examples. Each example had an average of 440 non-zero atomic features (stddev 123), so the *direct* representation of each example requires a minimum $440 \cdot \text{sizeof(int)} = 1760$ bytes, and the entire atomic feature matrix would require 1760 bytes · 1.65 million = 2.8 GB. Conversely, an indirectly represent inference requires no more 32 bytes: two floats (the cached confidence $h(i)$ and the bias term $b(i)$), a pointer to a tree cut (i), and a bool (the y-value $y(i)$). Indirectly storing the entire example set requires only 32 bytes · 1.65 million = 53 MB plus the treebank and tree cuts, a total of 400 MB in our implementation.

We used a sample size of $|S| = 100,000$ examples to build each decision tree, 16.5 times fewer than the entire example set. The dashed curve in Figure 1

[10] Bikel (2004) is a "clean room" reimplementation of the Collins (1999) model with comparable accuracy.

shows the percent of NP example weight lost due to sampling. As training progresses, fewer examples are informative to the model. Even though we ignore 94% of examples during feature selection, sampling loses less than 1% of the example weight after a day of training.

The NP classifier used in our final parser was an ensemble containing 2316 trees, which took five days to build. Overall, there were 96871 decision tree leaves, only 2339 of which were non-zero. There were an average of 40.4 (7.4 stddev) decision tree splits between the root of a tree and a non-zero leaf, and nearly all non-zero leaves were conjunctions of atomic feature *negations* (e.g. ¬(some child item is a verb) ∧ ¬(some child item is a preposition)). The non-zero leaf confidences were quite small in magnitude (0.107 mean, 0.069 stddev) but the training example margins over the entire ensemble were nonetheless quite high: 11.7 mean (2.92 stddev) for correct inferences, 30.6 mean (11.2 stddev) for incorrect inferences.

Table 2 profiles an NP training iteration, in which one decision tree is created and added to the NP ensemble. Feature selection in our algorithm (Steps 1.5–1.7) takes 1.5+38.2+127.6 = 167.3s, far faster than in naïve approaches. If we didn't do sampling but had 2.8GB to spare, we could eliminate the extraction step (Step 1.6) and instead cache the entire atomic feature matrix before the loop. However, tree building (Step 1.7) scales linearly in the number of examples, and would take $16.5 \cdot 127.6$s = 2105.4s using the entire example set. If we didn't do sampling and couldn't cache the atomic feature matrix, tree building would also require repeatedly performing extraction. The number of individual feature extractions needed to build a single decision tree is the sum over the internal nodes of the number of examples that percolate down to that node. There are an average of 40.8 (7.8 stddev) internal nodes in each tree and most of the examples fall in nearly all of them. This property is caused by the lopsided trees induced under ℓ_1 regularization. A conservative estimate is that each decision tree requires 25 extractions times the number of examples. So extraction would add at least $25 \cdot 16.5 \cdot 38.2$s = 15757.5*s* on top of 2105.40s, and hence building each decision tree would take at least $(15757.5+2105.40)/167.3 \approx$ *100 times* as long as it does currently.

Our decision tree ensembles contain over two orders of magnitude more compound features than those in Turian and Melamed (2005). Our overall training time was roughly equivalent to theirs. This ratio corroborates the above estimate.

5 Discussion

The NP classifier was trained only on the 1.65 million NP examples in the 9753 training sentences with ≤ 15 words (168.8 examples/sentence). The number of examples generated is quadratic in the sentence length, so there are 41.7 million NP examples in all 39832 training sentences of the whole Penn Treebank (1050 examples/sentence), 25 times as many as we are currently using.

The time complexity of each step in the training loop (Steps 1.5–11) is linear over the number of examples used by that step. When we scale up to the full treebank, feature selection will not require a sample 25 times larger, so it will no longer be the bottleneck in training. Instead, each iteration will be dominated by choosing leaf confidences and then updating the cached example confidences, which would require $25 \cdot (31.4\text{s} + 6.2\text{s}) = 940$s per iteration. These steps are crucial to the current training algorithm, because it is important to have example confidences that are current with respect to the model. Otherwise, we cannot determine the examples most poorly classified by the current model, and will have no basis for choosing an informative sample.

We might try to save training time by building *many* decision trees over a single sample and then updating the confidences of the entire example set using all the new trees. But, if this confidence update is done using feature tests, then we have merely deferred the cost of the confidence update over the entire example set. The amount of training done on a particular sample is proportional to the time subsequently spent updating confidences over the entire example set. To spend less time doing confidence updates, we must use a training regime that is *sub*-linear with respect to the training time. For example, Riezler (2004) reports that the ℓ_1 regularization term drives many of the model's parameters to zero during conjugate gradient optimization, which are

then pruned before subsequent optimization steps to avoid numerical instability. Instead of building decision tree(s) at each iteration, we could perform n-best feature selection followed by parallel optimization of the objective over the sample.

The main limitation of our work so far is that we can do training reasonably quickly only on short sentences, because a sentence with n words generates $O(n^2)$ training inferences in total. Although generating training examples in advance without a working parser (Sagae & Lavie, 2005) is much faster than using inference (Collins & Roark, 2004; Henderson, 2004; Taskar et al., 2004), our training time can probably be decreased further by choosing a parsing strategy with a lower branching factor. Like our work, Ratnaparkhi (1999) and Sagae and Lavie (2005) generate examples off-line, but their parsing strategies are essentially shift-reduce so each sentence generates only $O(n)$ training examples.

6 Conclusion

Our work has made advances in both accuracy and training speed of discriminative parsing. As far as we know, we present the first discriminative parser that surpasses a generative baseline on constituent parsing without using a generative component, and it does so with minimal linguistic cleverness.

The main bottleneck in our setting was memory. We could store the examples in memory only using an indirect representation. The most costly operation during training was accessing the features of a particular example from this indirect representation. We showed how to train a parser effectively under these conditions. In particular, we used principled sampling to estimate loss gradients and reduce the number of feature extractions. This approximation increased the speed of feature selection 100-fold.

We are exploring methods for scaling training up to larger example sets. We are also investigating the relationship between sample size, training time, classifier complexity, and accuracy. In addition, we shall make some standard improvements to our parser. Our parser should infer its own POS tags. A shift-reduce parsing strategy will generate fewer examples, and might lead to shorter training time. Lastly, we plan to give the model linguistically more sophisticated features. We also hope to apply the model to other structured learning tasks, such as syntax-driven SMT.

References

Bikel, D. M. (2004). Intricacies of Collins' parsing model. *Computational Linguistics*.

Black, E., Abney, S., Flickenger, D., Gdaniec, C., Grishman, R., Harrison, P., et al. (1991). A procedure for quantitatively comparing the syntactic coverage of English grammars. In *Speech and Natural Language*.

Bod, R. (2003). An efficient implementation of a new DOP model. In *EACL*.

Charniak, E., & Johnson, M. (2005). Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *ACL*.

Collins, M. (1999). *Head-driven statistical models for natural language parsing*. Doctoral dissertation.

Collins, M., & Roark, B. (2004). Incremental parsing with the perceptron algorithm. In *ACL*.

Collins, M., Schapire, R. E., & Singer, Y. (2002). Logistic regression, AdaBoost and Bregman distances. *Machine Learning*, *48*(1-3).

Duffield, N., Lund, C., & Thorup, M. (2005). *Priority sampling estimating arbitrary subset sums.* (`http://arxiv.org/abs/cs.DS/0509026`)

Henderson, J. (2004). Discriminative training of a neural network statistical parser. In *ACL*.

Klein, D., & Manning, C. D. (2001). Parsing and hypergraphs. In *IWPT*.

Kudo, T., Suzuki, J., & Isozaki, H. (2005). Boosting-based parse reranking with subtree features. In *ACL*.

Ng, A. Y. (2004). Feature selection, ℓ_1 vs. ℓ_2 regularization, and rotational invariance. In *ICML*.

Perkins, S., Lacker, K., & Theiler, J. (2003). Grafting: Fast, incremental feature selection by gradient descent in function space. *Journal of Machine Learning Research*, *3*.

Ratnaparkhi, A. (1996). A maximum entropy part-of-speech tagger. In *EMNLP*.

Ratnaparkhi, A. (1999). Learning to parse natural language with maximum entropy models. *Machine Learning*, *34*(1-3).

Riezler, S. (2004). Incremental feature selection of ℓ_1 regularization for relaxed maximum-entropy modeling. In *EMNLP*.

Sagae, K., & Lavie, A. (2005). A classifier-based parser with linear run-time complexity. In *IWPT*.

Schapire, R. E., & Singer, Y. (1999). Improved boosting using confidence-rated predictions. *Machine Learning*, *37*(3).

Smith, N. A., & Eisner, J. (2005). Contrastive estimation: Training log-linear models on unlabeled data. In *ACL*.

Taskar, B., Klein, D., Collins, M., Koller, D., & Manning, C. (2004). Max-margin parsing. In *EMNLP*.

Taylor, A., Marcus, M., & Santorini, B. (2003). The Penn Treebank: an overview. In A. Abeillé (Ed.), *Treebanks: Building and using parsed corpora* (chap. 1).

Turian, J., & Melamed, I. D. (2005). Constituent parsing by classification. In *IWPT*.

Turian, J., & Melamed, I. D. (2006). Advances in discriminative parsing. In *ACL*.