Proceedings of the 40th Annual Meeting of the Association for
Computational Linguistics (ACL), Philadelphia, July 2002, pp. 263-270.

# New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron

**Michael Collins**
AT&T Labs-Research,
Florham Park,
New Jersey.
mcollins@research.att.com

**Nigel Duffy**
iKuni Inc.,
3400 Hillview Ave., Building 5,
Palo Alto, CA 94304.
nigeduff@cs.ucsc.edu

## Abstract

This paper introduces new learning algorithms for natural language processing based on the perceptron algorithm. We show how the algorithms can be efficiently applied to exponential sized representations of parse trees, such as the “all subtrees” (DOP) representation described by (Bod 1998), or a representation tracking all sub-fragments of a tagged sentence. We give experimental results showing significant improvements on two tasks: parsing Wall Street Journal text, and named-entity extraction from web data.

## 1 Introduction

The perceptron algorithm is one of the oldest algorithms in machine learning, going back to (Rosenblatt 1958). It is an incredibly simple algorithm to implement, and yet it has been shown to be competitive with more recent learning methods such as support vector machines – see (Freund & Schapire 1999) for its application to image classification, for example.

This paper describes how the perceptron and voted perceptron algorithms can be used for parsing and tagging problems. Crucially, the algorithms can be efficiently applied to exponential sized representations of parse trees, such as the “all subtrees” (DOP) representation described by (Bod 1998), or a representation tracking all sub-fragments of a tagged sentence. It might seem paradoxical to be able to efficiently learn and apply a model with an exponential number of features.[1] The key to our algorithms is the “kernel” trick ((Cristianini and Shawe-Taylor 2000) discuss kernel methods at length). We describe how the inner product between feature vectors in these representations can be calculated efficiently using dynamic programming algorithms. This leads to polynomial time[2] algorithms for training and applying the perceptron. The kernels we describe are related to the kernels over discrete structures in (Haussler 1999; Lodhi et al. 2001).

A previous paper (Collins and Duffy 2001) showed improvements over a PCFG in parsing the ATIS task. In this paper we show that the method scales to far more complex domains. In parsing Wall Street Journal text, the method gives a 5.1% relative reduction in error rate over the model of (Collins 1999). In the second domain, detecting named-entity boundaries in web data, we show a 15.6% relative error reduction (an improvement in F-measure from 85.3% to 87.6%) over a state-of-the-art model, a maximum-entropy tagger. This result is derived using a new kernel, for tagged sequences, described in this paper. Both results rely on a new approach that incorporates the log-probability from a baseline model, in addition to the “all-fragments” features.

## 2 Feature–Vector Representations of Parse Trees and Tagged Sequences

This paper focuses on the task of choosing the correct parse or tag sequence for a sentence from a group of “candidates” for that sentence. The candidates might be enumerated by a number of methods. The experiments in this paper use the top $n$ candidates from a baseline probabilistic model: the model of (Collins 1999) for parsing, and a maximum-entropy tagger for named-entity recognition.

[1] Although see (Goodman 1996) for an efficient algorithm for the DOP model, which we discuss in section 7 of this paper.

[2] i.e., polynomial in the number of training examples, and the size of trees or sentences in training and test data.

The choice of representation is central: what features should be used as evidence in choosing between candidates? We will use a function $\mathbf{h}(\mathbf{x}) \in \mathbb{R}^d$ to denote a $d$-dimensional feature vector that represents a tree or tagged sequence $\mathbf{x}$. There are many possibilities for $\mathbf{h}(\mathbf{x})$. An obvious example for parse trees is to have one component of $\mathbf{h}(\mathbf{x})$ for each rule in a context-free grammar that underlies the trees. This is the representation used by Stochastic Context-Free Grammars. The feature vector tracks the counts of rules in the tree $\mathbf{x}$, thus encoding the sufficient statistics for the SCFG.

Given a representation, and two structures $\mathbf{x}$ and $\mathbf{y}$, the inner product between the structures can be defined as

$$\mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{y}) = \sum_{i=1}^{d} h_i(\mathbf{x}) h_i(\mathbf{y})$$

The idea of inner products between feature vectors is central to learning algorithms such as Support Vector Machines (SVMs), and is also central to the ideas in this paper. Intuitively, the inner product is a similarity measure between objects: structures with similar feature vectors will have high values for $\mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{y})$. More formally, it has been observed that many algorithms can be implemented using inner products between training examples alone, without direct access to the feature vectors themselves. As we will see in this paper, this can be crucial for the efficiency of learning with certain representations. Following the SVM literature, we call a function $K(\mathbf{x}, \mathbf{y})$ of two objects $\mathbf{x}$ and $\mathbf{y}$ a "kernel" if it can be shown that $K$ is an inner product in some feature space $\mathbf{h}$.

# 3 Algorithms

## 3.1 Notation

This section formalizes the idea of linear models for parsing or tagging. The method is related to the boosting approach to ranking problems (Freund et al. 1998), the Markov Random Field methods of (Johnson et al. 1999), and the boosting approaches for parsing in (Collins 2000). The set-up is as follows:

- Training data is a set of example input/output pairs. In parsing the training examples are $\{s_i, t_i\}$ where each $s_i$ is a sentence and each $t_i$ is the correct tree for that sentence.
- We assume some way of enumerating a set of candidates for a particular sentence. We use $\mathbf{x}_{ij}$ to denote the $j$'th candidate for the $i$'th sentence in training data, and $\mathcal{C}(s_i) = \{\mathbf{x}_{i1}, \mathbf{x}_{i2} \ldots\}$ to denote the set of candidates for $s_i$.
- Without loss of generality we take $\mathbf{x}_{i1}$ to be the correct candidate for $s_i$ (i.e., $\mathbf{x}_{i1} = t_i$).
- Each candidate $\mathbf{x}_{ij}$ is represented by a feature vector $\mathbf{h}(\mathbf{x}_{ij})$ in the space $\mathbb{R}^d$. The parameters of the model are also a vector $\mathbf{w} \in \mathbb{R}^d$. The output of the model on a training or test example $s$ is $\text{argmax}_{\mathbf{x} \in \mathcal{C}(s)} \mathbf{w} \cdot \mathbf{h}(\mathbf{x})$.

The key question, having defined a representation $\mathbf{h}$, is how to set the parameters $\mathbf{w}$. We discuss one method for setting the weights, the perceptron algorithm, in the next section.

## 3.2 The Perceptron Algorithm

Figure 1(a) shows the perceptron algorithm applied to the ranking task. The method assumes a training set as described in section 3.1, and a representation $\mathbf{h}$ of parse trees. The algorithm maintains a parameter vector $\mathbf{w}$, which is initially set to be all zeros. The algorithm then makes a pass over the training set, only updating the parameter vector when a mistake is made on an example. The parameter vector update is very simple, involving adding the difference of the offending examples' representations ($\mathbf{w} = \mathbf{w} + \mathbf{h}(\mathbf{x}_{i1}) - \mathbf{h}(\mathbf{x}_{ij})$ in the figure). Intuitively, this update has the effect of increasing the parameter values for features in the correct tree, and downweighting the parameter values for features in the competitor.

See (Cristianini and Shawe-Taylor 2000) for discussion of the perceptron algorithm, including an overview of various theorems justifying this way of setting the parameters. Briefly, the perceptron algorithm is guaranteed[3] to find a hyperplane that correctly classifies all training points, if such a hyperplane exists (i.e., the data is "separable"). Moreover, the number of mistakes made will be low, providing that the data is separable with "large margin", and

[3] To find such a hyperplane the algorithm must be run over the training set repeatedly until no mistakes are made. The algorithm in figure 1 includes just a single pass over the training set.

```
(a) **Define:**
    $F(\mathbf{x}) = \mathbf{w} \cdot \mathbf{h}(\mathbf{x})$.
    **Initialization:** Set parameters $\mathbf{w} = 0$
    **For** $i = 1 \ldots n$
      $j = \mathrm{argmax}_{j=1 \ldots n_i} F(\mathbf{x}_{ij})$
      **If** $(j \neq 1)$ **Then** $\mathbf{w} = \mathbf{w} + \mathbf{h}(\mathbf{x}_{i1}) - \mathbf{h}(\mathbf{x}_{ij})$
    **Output on test sentence** $s$**:**
          $\mathrm{argmax}_{\mathbf{x} \in \mathcal{C}(s)} F(\mathbf{x})$.
```

```
(b)**Define:**
    $G(\mathbf{x}) = \sum_{(i,j)} \alpha_{i,j} \left( \mathbf{h}(\mathbf{x}_{i1}) \cdot \mathbf{h}(\mathbf{x}) - \mathbf{h}(\mathbf{x}_{ij}) \cdot \mathbf{h}(\mathbf{x}) \right)$
    **Initialization:** Set dual parameters $\alpha_{i,j} = 0$
    **For** $i = 1 \ldots n$
      $j = \mathrm{argmax}_{j=1 \ldots n_i} G(\mathbf{x}_{ij})$
      **If** $(j \neq 1)$ **Then** $\alpha_{ij} = \alpha_{ij} + 1$
    **Output on test sentence** $s$**:**
          $\mathrm{argmax}_{\mathbf{x} \in \mathcal{C}(s)} G(\mathbf{x})$.
```

Figure 1: a) The perceptron algorithm for ranking problems. b) The algorithm in dual form.

this translates to guarantees about how the method generalizes to test examples. (Freund & Schapire 1999) give theorems showing that the voted perceptron (a variant described below) generalizes well even given non-separable data.

### 3.3 The Algorithm in Dual Form

Figure 1(b) shows an equivalent algorithm to the perceptron, an algorithm which we will call the "dual form" of the perceptron. The dual-form algorithm does not store a parameter vector $\mathbf{w}$, instead storing a set of dual parameters, $\alpha_{i,j}$ for $i = 1 \ldots n, j = 2 \ldots n_i$. The score for a parse $\mathbf{x}$ is defined by the dual parameters as

$$G(\mathbf{x}) = \sum_{(i,j)} \alpha_{i,j} \left( \mathbf{h}(\mathbf{x}_{i1}) \cdot \mathbf{h}(\mathbf{x}) - \mathbf{h}(\mathbf{x}_{ij}) \cdot \mathbf{h}(\mathbf{x}) \right)$$

This is in contrast to $F(\mathbf{x}) = \mathbf{w} \cdot \mathbf{h}(\mathbf{x})$, the score in the original algorithm.

In spite of these differences the algorithms give identical results on training and test examples: to see this, it can be verified that $\mathbf{w} = \sum_{i,j} \alpha_{i,j} \left( \mathbf{h}(\mathbf{x}_{i1}) - \mathbf{h}(\mathbf{x}_{ij}) \right)$, and hence that $G(\mathbf{x}) = F(\mathbf{x})$, throughout training.

The important difference between the algorithms lies in the analysis of their computational complexity. Say $T$ is the size of the training set, i.e., $T = \sum_i n_i$. Also, take $d$ to be the dimensionality of the parameter vector $\mathbf{w}$. Then the algorithm in figure 1(a) takes $O(Td)$ time.[4] This follows because $F(\mathbf{x})$ must be calculated for each member of the training set, and each calculation of $F$ involves $O(d)$ time. Now say the time taken to compute the inner product between two examples is $k$. The running time of the algorithm in figure 1(b) is $O(Tnk)$. This follows because throughout the algorithm the number of non-zero dual parameters is bounded by $n$, and hence the calculation of $G(\mathbf{x})$ takes at most $O(nk)$ time. (Note that the dual form algorithm runs in quadratic time in the number of training examples $n$, because $T \geq n$.)

The dual algorithm is therefore more efficient in cases where $nk << d$. This might seem unlikely to be the case – naively, it would be expected that the time to calculate the inner product $\mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{y})$ between two vectors to be at least $O(d)$. But it turns out that for some high-dimensional representations the inner product can be calculated in much better than $O(d)$ time, making the dual form algorithm more efficient than the original algorithm. The dual-form algorithm goes back to (Aizerman et al. 64). See (Cristianini and Shawe-Taylor 2000) for more explanation of the algorithm.

### 3.4 The Voted Perceptron

(Freund & Schapire 1999) describe a refinement of the perceptron algorithm, the "voted perceptron". They give theory which suggests that the voted perceptron is preferable in cases of noisy or unseparable data. The training phase of the algorithm is unchanged – the change is in how the method is applied to test examples. The algorithm in figure 1(b) can be considered to build a series of hypotheses $G^t(\mathbf{x})$, for $t = 1 \ldots n$, where $G^t$ is defined as

$$G^t(\mathbf{x}) = \sum_{(i \leq t, j)} \alpha_{i,j} \left( \mathbf{h}(\mathbf{x}_{i1}) \cdot \mathbf{h}(\mathbf{x}) - \mathbf{h}(\mathbf{x}_{ij}) \cdot \mathbf{h}(\mathbf{x}) \right)$$

$G^t$ is the scoring function from the algorithm trained on just the first $t$ training examples. The output of a model trained on the first $t$ examples for a sentence $s$

[4] If the vectors $\mathbf{h}(\mathbf{x})$ are sparse, then $d$ can be taken to be the number of non-zero elements of $\mathbf{h}$, assuming that it takes $O(d)$ time to add feature vectors with $O(d)$ non-zero elements, or to take inner products.

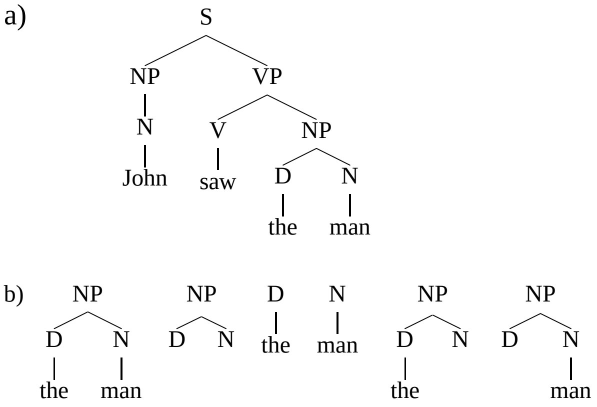

Figure 2: a) An example parse tree. b) The sub-trees of the NP covering *the man*. The tree in (a) contains all of these subtrees, as well as many others.

is $V^t(s) = \arg\max_{\mathbf{x}\in\mathcal{C}(s)} G^t(\mathbf{x})$. Thus the training algorithm can be considered to construct a sequence of $n$ models, $V^1 \ldots V^n$. On a test sentence $s$, each of these $n$ functions will return its own parse tree, $V^t(s)$ for $t = 1 \ldots n$. The voted perceptron picks the most likely tree as that which occurs most often in the set $\{V^1(s), V^2(s) \ldots V^n(s)\}$.

Note that $G^t$ is easily derived from $G^{t-1}$, through the identity $G^t(\mathbf{x}) = G^{t-1}(\mathbf{x}) + \sum_{j=2}^{n_t} \alpha_{t,j} \left(\mathbf{h}(\mathbf{x}_{t1}) \cdot \mathbf{h}(\mathbf{x}) - \mathbf{h}(\mathbf{x}_{tj}) \cdot \mathbf{h}(\mathbf{x})\right)$. Because of this the voted perceptron can be implemented with the same number of kernel calculations, and hence roughly the same computational complexity, as the original perceptron.

## 4 A Tree Kernel

We now consider a representation that tracks all subtrees seen in training data, the representation studied extensively by (Bod 1998). See figure 2 for an example. Conceptually we begin by enumerating all tree fragments that occur in the training data $1 \ldots d$. Note that this is done only implicitly. Each tree is represented by a $d$ dimensional vector where the $i$'th component counts the number of occurences of the $i$'th tree fragment. Define the function $h_i(\mathbf{x})$ to be the number of occurences of the $i$'th tree fragment in tree $\mathbf{x}$, so that $\mathbf{x}$ is now represented as $\mathbf{h}(\mathbf{x}) = (h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_d(\mathbf{x}))$. Note that $d$ will be huge (a given tree will have a number of subtrees that is exponential in its size). Because of this we aim to design algorithms whose computational complexity is independent of $d$.

The key to our efficient use of this representation is a dynamic programming algorithm that computes the inner product between two examples $\mathbf{x}_1$ and $\mathbf{x}_2$ in polynomial (in the size of the trees involved), rather than $O(d)$, time. The algorithm is described in (Collins and Duffy 2001), but for completeness we repeat it here. We first define the set of nodes in trees $\mathbf{x}_1$ and $\mathbf{x}_2$ as $N_1$ and $N_2$ respectively. We define the indicator function $I_i(n)$ to be 1 if sub-tree $i$ is seen rooted at node $n$ and 0 otherwise. It follows that $h_i(\mathbf{x}_1) = \sum_{n_1 \in N_1} I_i(n_1)$ and $h_i(\mathbf{x}_2) = \sum_{n_2 \in N_2} I_i(n_2)$. The first step to efficient computation of the inner product is the following property:

$$\begin{aligned} \mathbf{h}(\mathbf{x}_1) \cdot \mathbf{h}(\mathbf{x}_2) &= \textstyle\sum_i h_i(\mathbf{x}_1) h_i(\mathbf{x}_2) \\ &= \textstyle\sum_i \left(\sum_{n_1 \in N_1} I_i(n_1)\right) \left(\sum_{n_2 \in N_2} I_i(n_2)\right) \\ &= \textstyle\sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \sum_i I_i(n_1) I_i(n_2) \\ &= \textstyle\sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \Delta(n_1, n_2) \end{aligned}$$

where we define $\Delta(n_1, n_2) = \sum_i I_i(n_1) I_i(n_2)$. Next, we note that $\Delta(n_1, n_2)$ can be computed efficiently, due to the following recursive definition:

- If the productions at $n_1$ and $n_2$ are different $\Delta(n_1, n_2) = 0$.
- If the productions at $n_1$ and $n_2$ are the same, and $n_1$ and $n_2$ are pre-terminals, then $\Delta(n_1, n_2) = 1$.[5]
- Else if the productions at $n_1$ and $n_2$ are the same and $n_1$ and $n_2$ are not pre-terminals,

$$\Delta(n_1, n_2) = \prod_{j=1}^{nc(n_1)} \left(1 + \Delta\left(ch(n_1, j), ch(n_2, j)\right)\right) ,$$

where $nc(n_1)$ is the number of children of $n_1$ in the tree; because the productions at $n_1$/$n_2$ are the same, we have $nc(n_1) = nc(n_2)$. The $i$'th child-node of $n_1$ is $ch(n_1, i)$.

To see that this recursive definition is correct, note that $\Delta(n_1, n_2) = \sum_i I_i(n_1) I_i(n_2)$ simply counts the number of *common subtrees* that are found rooted at both $n_1$ and $n_2$. The first two cases are trivially correct. The last, recursive, definition follows because a common subtree for $n_1$ and $n_2$ can be formed by taking the production at $n_1$/$n_2$, together with a choice at each child of simply taking the non-terminal at that child, or any one of the common sub-trees at that child. Thus there are

[5] Pre-terminals are nodes directly above words in the surface string, for example the `N, V`, and `D` symbols in Figure 2.

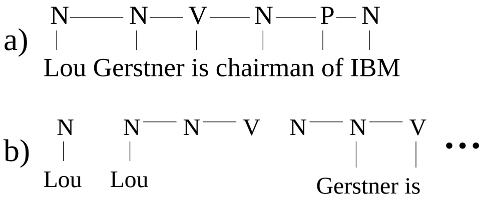

Figure 3: a) A tagged sequence. b) Example "fragments" of the tagged sequence: the tagging kernel is sensitive to the counts of all such fragments.

$(1+\Delta(child(n_1,i),child(n_2,i)))$ possible choices at the $i$'th child. (Note that a similar recursion is described by Goodman (Goodman 1996), Goodman's application being the conversion of Bod's model (Bod 1998) to an equivalent PCFG.)

It is clear from the identity $\mathbf{h}(\mathbf{x}_1)\cdot\mathbf{h}(\mathbf{x}_2) = \sum_{n_1,n_2}\Delta(n_1,n_2)$, and the recursive definition of $\Delta(n_1,n_2)$, that $\mathbf{h}(\mathbf{x}_1)\cdot\mathbf{h}(\mathbf{x}_2)$ can be calculated in $O(|N_1||N_2|)$ time: the matrix of $\Delta(n_1,n_2)$ values can be filled in, then summed.[6]

Since there will be many more tree fragments of larger size – say depth four versus depth three – it makes sense to downweight the contribution of larger tree fragments to the kernel. This can be achieved by introducing a parameter $0 < \lambda \leq 1$, and modifying the base case and recursive case of the definitions of $\Delta$ to be respectively $\Delta(n_1,n_2) = \lambda$ and $\Delta(n_1,n_2) = \lambda\prod_{j=1}^{nc(n_1)}(1+\Delta(ch(n_1,j),ch(n_2,j)))$. This corresponds to a modified kernel $\mathbf{h}(\mathbf{x}_1)\cdot\mathbf{h}(\mathbf{x}_2) = \sum_i \lambda^{size_i} h_i(\mathbf{x}_1)h_i(\mathbf{x}_2)$ where $size_i$ is the number of rules in the $i$'th fragment. This is roughly equivalent to having a prior that large sub-trees will be less useful in the learning task.

## 5 A Tagging Kernel

The second problem we consider is tagging, where each word in a sentence is mapped to one of a finite set of tags. The tags might represent part-of-speech tags, named-entity boundaries, base noun-phrases, or other structures. In the experiments in this paper we consider named-entity recognition.

A tagged sequence is a sequence of word/state pairs $\mathbf{x} = \{w_1/s_1 \ldots w_n/s_n\}$ where $w_i$ is the $i$'th word, and $s_i$ is the tag for that word. The particular representation we consider is similar to the all sub-trees representation for trees. A tagged-sequence "fragment" is a subgraph that contains a subsequence of state labels, where each label may or may not contain the word below it. See figure 3 for an example. Each tagged sequence is represented by a $d$ dimensional vector where the $i$'th component $h_i(\mathbf{x})$ counts the number of occurrences of the $i$'th fragment in $\mathbf{x}$.

The inner product under this representation can be calculated using dynamic programming in a very similar way to the tree algorithm. We first define the set of states in tagged sequences $\mathbf{x}_1$ and $\mathbf{x}_2$ as $N_1$ and $N_2$ respectively. Each state has an associated *label* and an associated *word*. We define the indicator function $I_i(n)$ to be 1 if fragment $i$ is seen with left-most state at node $n$, and 0 otherwise. It follows that $h_i(\mathbf{x}_1) = \sum_{n_1\in N_1} I_i(n_1)$ and $h_i(\mathbf{x}_2) = \sum_{n_2\in N_2} I_i(n_2)$. As before, some simple algebra shows that

$$\mathbf{h}(\mathbf{x}_1)\cdot\mathbf{h}(\mathbf{x}_2) = \sum_{n_1\in N_1}\sum_{n_2\in N_2}\Delta(n_1,n_2)$$

where we define $\Delta(n_1,n_2) = \sum_i I_i(n_1)I_i(n_2)$. Next, for any given state $n_1 \in N_1$ define $next(n_1)$ to be the state to the right of $n_1$ in the structure $\mathbf{x}_1$. An analogous definition holds for $next(n_2)$. Then $\Delta(n_1,n_2)$ can be computed using dynamic programming, due to a recursive definition:

- If the state labels at $n_1$ and $n_2$ are different $\Delta(n_1,n_2) = 0$.
- If the state labels at $n_1$ and $n_2$ are the same, but the words at $n_1$ and $n_2$ are different, then $\Delta(n_1,n_2) = 1+\Delta(next(n_1),next(n_2))$.
- Else if the state labels at $n_1$ and $n_2$ are the same, and the words at $n_1$ and $n_2$ are the same, then $\Delta(n_1,n_2) = 2+2\times\Delta(next(n_1),next(n_2))$.

There are a couple of useful modifications to this kernel. One is to introduce a parameter $0 < \lambda \leq 1$ which penalizes larger substructures. The recursive definitions are modfied to be $\Delta(n_1,n_2) = 1+\lambda\Delta(next(n_1),next(n_2))$ and $\Delta(n_1,n_2) = 2+2\lambda\Delta(next(n_1),next(n_2))$ respectively. This gives an inner product $\sum_i \lambda^{size_i} h_i(\mathbf{x}_1)h_i(\mathbf{x}_2)$ where $size_i$ is the number of state labels in the $i$th fragment.

Another useful modification is as follows. Define

[6] This can be a pessimistic estimate of the runtime. A more useful characterization is that it runs in time linear in the number of members $(n_1,n_2)\in N_1\times N_2$ such that the productions at $n_1$ and $n_2$ are the same. In our data we have found the number of nodes with identical productions to be approximately linear in the size of the trees, so the running time is also close to linear in the size of the trees.

| MODEL | ≤ 40 Words (2245 sentences) | | | | |
|---|---|---|---|---|---|
| | LR | LP | CBs | 0 CBs | 2 CBs |
| CO99 | 88.5% | 88.7% | 0.92 | 66.7% | 87.1% |
| VP | 89.1% | 89.4% | 0.85 | 69.3% | 88.2% |

| MODEL | ≤ 100 Words (2416 sentences) | | | | |
|---|---|---|---|---|---|
| | LR | LP | CBs | 0 CBs | 2 CBs |
| CO99 | 88.1% | 88.3% | 1.06 | 64.0% | 85.1% |
| VP | 88.6% | 88.9% | 0.99 | 66.5% | 86.3% |

Figure 4: Results on Section 23 of the WSJ Treebank. **LR/LP** = labeled recall/precision. **CBs** = average number of crossing brackets per sentence. **0 CBs,** 2 **CBs** are the percentage of sentences with 0 or $\leq 2$ crossing brackets respectively. CO99 is model 2 of (Collins 1999). VP is the voted perceptron with the tree kernel.

$Sim_1(w_1, w_2)$ for words $w_1$ and $w_2$ to be 1 if $w_1 = w_2$, 0 otherwise. Define $Sim_2(w_1, w_2)$ to be 1 if $w_1$ and $w_2$ share the same *word features*, 0 otherwise. For example, $Sim_2$ might be defined to be 1 if $w_1$ and $w_2$ are both capitalized: in this case $Sim_2$ is a looser notion of similarity than the exact match criterion of $Sim_1$. Finally, the definition of $\Delta$ can be modified to:

- If labels at $n_1/n_2$ are different, $\Delta(n_1, n_2) = 0$.
- Else $\Delta(n_1, n_2) =$
  $(1 + 0.5 Sim_1(w_1, w_2) + 0.5 Sim_2(w_1, w_2))$
  $\times (1 + \lambda \times \Delta(next(n_1), next(n_2)))$

where $w_1$, $w_2$ are the words at $n_1$ and $n_2$ respectively. This inner product implicitly includes features which track word features, and thus can make better use of sparse data.

## 6 Experiments

### 6.1 Parsing Wall Street Journal Text

We used the same data set as that described in (Collins 2000). The Penn Wall Street Journal treebank (Marcus et al. 1993) was used as training and test data. Sections 2-21 inclusive (around 40,000 sentences) were used as training data, section 23 was used as the final test set. Of the 40,000 training sentences, the first 36,000 were used to train the perceptron. The remaining 4,000 sentences were used as development data, and for tuning parameters of the algorithm. Model 2 of (Collins 1999) was used to parse both the training and test data, producing multiple hypotheses for each sentence. In order to gain a representative set of training data, the 36,000 training sentences were parsed in 2,000 sentence chunks, each chunk being parsed with a model trained on the remaining 34,000 sentences (this prevented the initial model from being unrealistically "good" on the training sentences). The 4,000 development sentences were parsed with a model trained on the 36,000 training sentences. Section 23 was parsed with a model trained on all 40,000 sentences.

The representation we use incorporates the probability from the original model, as well as the all-subtrees representation. We introduce a parameter $\beta$ which controls the relative contribution of the two terms. If $L(\mathbf{x})$ is the log probability of a tree $\mathbf{x}$ under the original probability model, and $\mathbf{h}(\mathbf{x}) = (h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_d(\mathbf{x}))$ is the feature vector under the all subtrees representation, then the new representation is $\mathbf{h}_2(\mathbf{x}) = (\sqrt{\beta}L(\mathbf{x}), h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_d(\mathbf{x}))$, and the inner product between two examples $\mathbf{x}$ and $\mathbf{y}$ is $\mathbf{h}_2(\mathbf{x}) \cdot \mathbf{h}_2(\mathbf{y}) = \beta L(\mathbf{x})L(\mathbf{y}) + \mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{y})$. This allows the perceptron algorithm to use the probability from the original model as well as the subtrees information to rank trees. We would thus expect the model to do at least as well as the original probabilistic model.

The algorithm in figure 1(b) was applied to the problem, with the inner product $\mathbf{h}_2(\mathbf{x}) \cdot \mathbf{h}_2(\mathbf{y})$ used in the definition of $G(\mathbf{x})$. The algorithm in 1(b) runs in approximately quadratic time in the number of training examples. This made it somewhat expensive to run the algorithm over all 36,000 training sentences in one pass. Instead, we broke the training set into 6 chunks of roughly equal size, and trained 6 separate perceptrons on these data sets. This has the advantage of reducing training time, both because of the quadratic dependence on training set size, and also because it is easy to train the 6 models in parallel. The outputs from the 6 runs on test examples were combined through the voting procedure described in section 3.4.

Figure 4 shows the results for the voted perceptron with the tree kernel. The parameters $\beta$ and $\lambda$ were set to 0.2 and 0.3 respectively through tuning on the development set. The method shows a 0.6% absolute improvement in average precision and recall (from 88.2% to 88.8% on sentences $\leq$ 100 words), a 5.1% relative reduction in error. The boosting method of (Collins 2000) showed 89.6%/89.9% recall and precision on reranking approaches for the same datasets (sentences less than 100 words in length). (Charniak 2000) describes a different method which achieves very similar performance to (Collins 2000). (Bod 2001) describes experiments giving 90.6%/90.8% recall and precision for sentences of less than 40 words in length, using the all-subtrees representation, but using very different algorithms and parameter estimation methods from the perceptron algorithms in this paper (see section 7 for more discussion).

### 6.2 Named–Entity Extraction

Over a period of a year or so we have had over one million words of named-entity data annotated. The data is drawn from web pages, the aim being to support a question-answering system over web data. A number of categories are annotated: the usual people, organization and location categories, as well as less frequent categories such as brand-names, scientific terms, event titles (such as concerts) and so on. As a result, we created a training set of 53,609 sentences (1,047,491 words), and a test set of 14,717 sentences (291,898 words).

The task we consider is to recover named-entity boundaries. We leave the recovery of the categories of entities to a separate stage of processing. We evaluate different methods on the task through precision and recall.[7] The problem can be framed as a tagging task – to tag each word as being either the start of an entity, a continuation of an entity, or not to be part of an entity at all. As a baseline model we used a maximum entropy tagger, very similar to the one described in (Ratnaparkhi 1996). Maximum entropy taggers have been shown to be highly competitive on a number of tagging tasks, such as part-of-speech tagging (Ratnaparkhi 1996), and named-entity recognition (Borthwick et. al 1998). Thus the maximum-entropy tagger we used represents a serious baseline for the task. We used a feature set which included the current, next, and previous word; the previous two tags; various capitalization and other features of the word being tagged (the full feature set is described in (Collins 2002a)).

As a baseline we trained a model on the full 53,609 sentences of training data, and decoded the 14,717 sentences of test data using a beam search which keeps the top 20 hypotheses at each stage of a left-to-right search. In training the voted perceptron we split the training data into a 41,992 sentence training set, and a 11,617 sentence development set. The training set was split into 5 portions, and in each case the maximum-entropy tagger was trained on 4/5 of the data, then used to decode the remaining 1/5. In this way the whole training data was decoded. The top 20 hypotheses under a beam search, together with their log probabilities, were recovered for each training sentence. In a similar way, a model trained on the 41,992 sentence set was used to produce 20 hypotheses for each sentence in the development set.

|  | P | R | F |
|---|---|---|---|
| Max-Ent | 84.4% | 86.3% | 85.3% |
| Perc. | 86.1% | 89.1% | 87.6% |
| Imp. | 10.9% | 20.4% | 15.6% |

Figure 5: Results for the max-ent and voted perceptron methods. "Imp." is the relative error reduction given by using the perceptron. $P$ = precision, $R$ = recall, $F$ = F-measure.

As in the parsing experiments, the final kernel incorporates the probability from the maximum entropy tagger, i.e. $\mathbf{h}_2(\mathbf{x}) \cdot \mathbf{h}_2(\mathbf{y}) = \beta L(\mathbf{x})L(\mathbf{y}) + \mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{y})$ where $L(\mathbf{x})$ is the log-likelihood of $\mathbf{x}$ under the tagging model, $\mathbf{h}(\mathbf{x}) \cdot \mathbf{h}(\mathbf{y})$ is the tagging kernel described previously, and $\beta$ is a parameter weighting the two terms. The other free parameter in the kernel is $\lambda$, which determines how quickly larger structures are downweighted. In running several training runs with different parameter values, and then testing error rates on the development set, the best parameter values we found were $\beta = 0.2$, $\lambda = 0.5$. Figure 5 shows results on the test data for the baseline maximum-entropy tagger, and the voted perceptron. The results show a 15.6% relative improvement in F-measure.

## 7 Relationship to Previous Work

(Bod 1998) describes quite different parameter estimation and parsing methods for the DOP representation. The methods explicitly deal with the parameters associated with subtrees, with sub-sampling of tree fragments making the computation manageable. Even after this, Bod's method is left with a huge grammar: (Bod 2001) describes a grammar with

[7] If a method proposes $p$ entities on the test set, and $c$ of these are correct then the precision of a method is $100\% * c/p$. Similarly, if $g$ is the number of entities in the human annotated version of the test set, then the recall is $100\% * c/g$.

over 5 million sub-structures. The method requires search for the 1,000 most probable derivations under this grammar, using beam search, presumably a challenging computational task given the size of the grammar. In spite of these problems, (Bod 2001) gives excellent results for the method on parsing Wall Street Journal text. The algorithms in this paper have a different flavor, avoiding the need to explicitly deal with feature vectors that track all subtrees, and also avoiding the need to sum over an exponential number of derivations underlying a given tree.

(Goodman 1996) gives a polynomial time conversion of a DOP model into an equivalent PCFG whose size is linear in the size of the training set. The method uses a similar recursion to the common sub-trees recursion described in this paper. Goodman's method still leaves exact parsing under the model intractable (because of the need to sum over multiple derivations underlying the same tree), but he gives an approximation to finding the most probable tree, which can be computed efficiently.

From a theoretical point of view, it is difficult to find motivation for the parameter estimation methods used by (Bod 1998) – see (Johnson 2002) for discussion. In contrast, the parameter estimation methods in this paper have a strong theoretical basis (see (Cristianini and Shawe-Taylor 2000) chapter 2 and (Freund & Schapire 1999) for statistical theory underlying the perceptron).

For related work on the voted perceptron algorithm applied to NLP problems, see (Collins 2002a) and (Collins 2002b). (Collins 2002a) describes experiments on the same named-entity dataset as in this paper, but using explicit features rather than kernels. (Collins 2002b) describes how the voted perceptron can be used to train maximum-entropy style taggers, and also gives a more thorough discussion of the theory behind the perceptron algorithm applied to ranking tasks.

**Acknowledgements** Many thanks to Jack Minisi for annotating the named-entity data used in the experiments. Thanks to Rob Schapire and Yoram Singer for many useful discussions.

## References

Aizerman, M., Braverman, E., & Rozonoer, L. (1964). Theoretical Foundations of the Potential Function Method in Pattern Recognition Learning. In *Automation and Remote Control*, 25:821–837.

Bod, R. (1998). *Beyond Grammar: An Experience-Based Theory of Language*. CSLI Publications/Cambridge University Press.

Bod, R. (2001). What is the Minimal Set of Fragments that Achieves Maximal Parse Accuracy? In *Proceedings of ACL 2001*.

Borthwick, A., Sterling, J., Agichtein, E., and Grishman, R. (1998). Exploiting Diverse Knowledge Sources via Maximum Entropy in Named Entity Recognition. *Proc. of the Sixth Workshop on Very Large Corpora*.

Charniak, E. (2000). A maximum-entropy-inspired parser. In *Proceedings of NAACL 2000.*

Collins, M. 1999. Head-Driven Statistical Models for Natural Language Parsing. PhD Dissertation, University of Pennsylvania.

Collins, M. (2000). Discriminative Reranking for Natural Language Parsing. *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000).*

Collins, M., and Duffy, N. (2001). Convolution Kernels for Natural Language. In *Proceedings of Neural Information Processing Systems (NIPS 14).*

Collins, M. (2002a). Ranking Algorithms for Named–Entity Extraction: Boosting and the Voted Perceptron. In *Proceedings of ACL 2002.*

Collins, M. (2002b). Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with the Perceptron Algorithm. In *Proceedings of EMNLP 2002.*

Cristianini, N., and Shawe-Tayor, J. (2000). *An introduction to Support Vector Machines and other kernel-based learning methods.* Cambridge University Press.

Freund, Y. & Schapire, R. (1999). Large Margin Classification using the Perceptron Algorithm. In *Machine Learning,* 37(3):277–296.

Freund, Y., Iyer, R.,Schapire, R.E., & Singer, Y. (1998). An efficient boosting algorithm for combining preferences. In *Machine Learning: Proceedings of the Fifteenth International Conference*. San Francisco: Morgan Kaufmann.

Goodman, J. (1996). Efficient algorithms for parsing the DOP model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing,* pages 143-152.

Haussler, D. (1999). *Convolution Kernels on Discrete Structures.* Technical report, University of Santa Cruz.

Johnson, M., Geman, S., Canon, S., Chi, S., & Riezler, S. (1999). Estimators for stochastic 'unification-based" grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*.

Johnson, M. (2002). The DOP estimation method is biased and inconsistent. *Computational Linguistics, 28*, 71-76.

Lodhi, H., Christianini, N., Shawe-Taylor, J., & Watkins, C. (2001). Text Classification using String Kernels. In *Advances in Neural Information Processing Systems 13*, MIT Press.

Marcus, M., Santorini, B., & Marcinkiewicz, M. (1993). Building a large annotated corpus of english: The Penn treebank. *Computational Linguistics, 19*, 313-330.

Ratnaparkhi, A. (1996). A maximum entropy part-of-speech tagger. In *Proceedings of the empirical methods in natural language processing conference*.

Rosenblatt, F. 1958. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65, 386–408. (Reprinted in *Neurocomputing* (MIT Press, 1998).)