Parsing with a Single Neuron: Convolution Kernels for Natural Language Problems

Michael Collins† and Nigel Duffy‡

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

‡Department of Computer Science,
University of California at Santa Cruz.
nigeduff@cs.ucsc.edu

Abstract

This paper introduces new training criteria and algorithms for NLP problems, based on the Support Vector Machine (SVM) approach to classification problems. SVMs can be applied efficiently to a feature vector representation ϕ as long as the inner product between feature vectors can be computed efficiently. We show how this allows SVMs to be applied to representations that would be intractable, or certainly challenging, for other methods.

1 Introduction

Recently, methods such as Markov Random Fields (MRFs) (Abney 1997; Della Pietra et al. 1997; Johnson et al. 1999) and Boosting (Abney et al. 1999; Collins 2000) have been introduced to natural language problems. These methods have the advantage of being highly flexible, in that they allow the objects being modeled to be represented as arbitrary feature vectors. The motivation is that the freedom to use richer representations will lead to improved performance on NLP tasks: there is empirical evidence for this, in improvements on WSJ treebank parsing through additional features (Charniak 1999; Collins 2000) or in the successful application of MRFs to linguistically motivated representations such as unification grammars or LFG parses (Abney 1997; Johnson et al. 1999).

Our aim in this paper is to introduce new learning algorithms which take advantage of rich, high dimensional representations of NLP objects. (Cortes & Vapnik 1995) describe two major problems when working with such rich representations. The first is "technical": how can models be trained and applied efficiently in high dimensional spaces? The second is "conceptual": which training criteria lead to good generalization (i.e., avoid problems with overfitting) in spite of the model having a large number of parameters?

Support Vector Machines (SVMs) (Cortes & Vapnik 1995; Burges 1998) have been proposed as a classification method which takes advantage of rich feature spaces, while avoiding computational problems and problems of overtraining. The use of kernels allows learning to be performed efficiently in high dimensional spaces; moreover theoretical results (Bartlett & Shawe–Taylor 1998) show that large margin classifiers can generalize well in large or even infinite-dimensional feature spaces. Unlike other approaches such as neural networks, SVM training algorithms involve the optimization of a loss function with a unique minimum, avoiding problems of local minima. Empirically, SVMs have been shown to be highly competitive on tasks such as digit recognition (Cortes & Vapnik 1995) or text classification (Joachims 1998). SVMs construct a separating hyperplane, or a single "neuron", in the feature space (hence the title of this paper).

The first part of this paper introduces new training criteria and algorithms for NLP problems, based on SVMs. The SVM technique bears some similarity to boosting and MRFs, but there is a crucial difference. All of these methods represent a given NLP object $\mathbf{x}$ (e.g., a parse tree) as a feature vector $\phi(\mathbf{x})$. SVMs can be applied to a feature vector representation ϕ as long as the inner product[1] $\langle\phi(\mathbf{x}), \phi(\mathbf{y})\rangle$ between feature vectors can be computed efficiently – irrespective of the dimensionality of the feature vectors ϕ. This property has been used to great effect in classification tasks, allowing models to be trained and applied in feature spaces which are very high dimensional, but where inner products can nevertheless be computed efficiently.

Most previous work on SVMs has focused on tasks where the objects to be classified are vectors in Euclidean space, rather than the discrete objects commonly found in NLP problems. The second part of this paper describes various feature vector representations for structures such as trees, dependency structures and paired sentences/tag sequences. In each case we show that inner products between feature vectors can be computed in polynomial time, in spite of exponentially sized representations (such as a feature vector tracking all sub-trees of a parse tree). The inner products are convolution kernels over discrete structures, as proposed by (Haussler 1999). These representations could in principle be used in other learning frameworks such as boosting and MRF methods, but in practice would be computationally prohibitive due to the exponential size of feature vectors[2]. In section 4 we give some initial experiments as "proof of concept" of the approach.

[1]We use $\langle\mathbf{x}, \mathbf{y}\rangle$ to denote the inner product (dot product $\mathbf{x} \cdot \mathbf{y}$) between vectors $\mathbf{x}$ and $\mathbf{y}$.

[2]Although (Bod 1998) has previously applied the "all sub-trees" representation – see section 5.1 for discussion.

1.1 Linear Models for NLP Problems

This section describes the problems we are considering in this paper. We will use the parsing problem throughout as an example, although the framework is also relevant to tagging tasks such as part-of-speech tagging, named entity tagging or to problems such as speech recognition. The set-up is as follows:

- Training data is a set of example input/output pairs; in parsing the examples are pairs $\{s_i, t_i\}$ 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. A context-free grammar taken straight from the training examples themselves is one way of enumerating candidates. Another choice is to use a hand-crafted grammar (such as the LFG grammar in (Johnson et al. 1999)) or to take the n most probable parses from an existing parser (as in (Collins 2000)).
- Without loss of generality we take $\mathbf{x}_{i1}$ to be the correct parse for s_i (i.e., $\mathbf{x}_{i1} = t_i$).
- Each candidate $\mathbf{x}_{ij}$ is represented by a feature vector $\phi(\mathbf{x}_{ij})$ in the space $\mathbb{R}^n$. The parameters of the model are also a vector $\bar{w} \in \mathbb{R}^n$. We then define the "ranking score" of each example as

$$F(\mathbf{x}_{ij}) = \langle\bar{w}, \phi(\mathbf{x}_{ij})\rangle$$

This score is interpreted as an indication of the plausibility of the candidate; in particular, the output of the model on a training or test example s is

$$\operatorname*{argmax}_{\mathbf{x} \in \mathcal{C}(s)} \langle\bar{w}, \phi(\mathbf{x})\rangle$$

The representation is taken to be fixed: training the model involves setting the value of the parameter vector $\bar{w}$.

This framework is general enough to include several approaches. A closely related approach to the one in this paper is the use of Markov Random Fields (MRFs) by (Johnson et al. 1999). An LFG grammar is used to enumerate a set of candidate features for each sentence. A conditional probability is defined for each of the candidates as

$$P(\mathbf{x}_{ij}|s_i) = \frac{e^{\langle \bar{w}, \phi(\mathbf{x}_{ij}) \rangle}}{\sum_{z \in \mathcal{C}(s_i)} e^{\langle \bar{w}, \phi(z) \rangle}}$$

and the output of the model on a training or test example s is

$$\operatorname*{argmax}_{\mathbf{x} \in \mathcal{C}(s)} P(\mathbf{x}|s) = \operatorname*{argmax}_{\mathbf{x} \in \mathcal{C}(s)} \langle \bar{w}, \phi(\mathbf{x}) \rangle$$

The parameters are set to maximize the following objective function, which is a combination of the log-likelihood of the training data and a Gaussian prior on the parameter values:

$$\sum_i \log P(\mathbf{x}_{i1}|s_i) - ||\bar{w}||^2$$

This function combines an incentive to raise the probability of each correct parse close to 1 (and hence to push $F(\mathbf{x}_{i1}) >> F(\mathbf{x}_{ij})$ for all $i, j \geq 2$) with a penalty for large parameter values. As we will see, this bears some similarities to the training criterion for SVMs.

2 Support Vector Machines

2.1 Maximum Margin Hyperplanes

We now discuss the SVM training criterion for the problem type described in the previous section (the method is derived by a transformation from ranking problems to a margin-based classification problem as described in (Freund et al. 1998)). The question is how to train the parameters $\bar{w}$ of a linear model. First, note that a ranking function that correctly ranked the correct parse above all competing candidates would satisfy the conditions $F(\mathbf{x}_{i1}) > F(\mathbf{x}_{ij})$ or equivalently $\langle \bar{w}, \phi(\mathbf{x}_{i1}) - \phi(\mathbf{x}_{ij}) \rangle > 0$ for all i, $j \geq 2$.

A central concept in Support Vector Machines is the *margin* of a hyperplane on a given training set. The margin of a hyperplane is the minimal distance of the hyperplane to any of the training data points. The distance from the hyperplane $\bar{w}$ to a point $(\phi(\mathbf{x}_{i1}) - \phi(\mathbf{x}_{ij}))$ is $\langle \bar{w}, \phi(\mathbf{x}_{i1}) - \phi(\mathbf{x}_{ij}) \rangle / ||\bar{w}||$. The margin of a hyperplane $\bar{w}$ is therefore

$$\min_{i,j \geq 2} \frac{\langle \bar{w}, \phi(\mathbf{x}_{i1}) - \phi(\mathbf{x}_{ij}) \rangle}{||\bar{w}||}$$

The SVM training criterion is to choose the hyperplane which correctly classifies all training examples, and has maximum margin. This is equivalent (Cortes & Vapnik 1995) to defining the optimal hyperplane $\bar{w}^*$ as the $\bar{w}$ which minimizes $||\bar{w}||^2$ subject to the constraints

$$\langle \bar{w}, \phi(\mathbf{x}_{i1}) - \phi(\mathbf{x}_{ij}) \rangle \geq 1$$

for all i, $j \geq 2$. Search for the optimal parameter values $\bar{w}^*$ is a quadratic programming problem, see (Platt 1998) for a discussion of optimization algorithms for SVMs.

In some cases it may not be possible (or desirable) to classify all points in training data correctly. For this reason, *slack variables* (Cortes & Vapnik 1995) $\epsilon_{i,j}$ can be introduced, the modified objective function being to minimize

$$||\bar{w}||^2 + C \sum_{(i,j)} \epsilon_{ij} \tag{1}$$

(we use $\sum_{(i,j)}$ as shorthand for $\sum_i \sum_{j=2}^{n_i}$) subject to the constraints that

$$\langle \bar{w}, \phi(\mathbf{x}_{i1}) - \phi(\mathbf{x}_{ij}) \rangle \geq 1 - \epsilon_{ij} \tag{2}$$

for all i, $j \geq 2$. C is a constant that can be optimized through cross-validation. On some (usually most) training examples, $\epsilon_{i,j}$ will be zero and the margin will be greater than 1. On others, $\epsilon_{i,j}$ will be greater than zero: when $\epsilon_{i,j}$ is greater than 1 the i, j'th training example is incorrectly classified.

2.2 Representation using Kernels

A key point (Kimeldorf & Wahba 1971; Cortes & Vapnik 1995) is that the maximum margin hyperplane can always be expressed as a linear combination of the training examples themselves (recall that both $\bar{w}$ and $\phi(\mathbf{x})$ are in the same space $\mathbb{R}^n$):

$$\bar{w}^* = \sum_{(i,j)} \alpha_{i,j} \left(\phi(\mathbf{x}_{i1}) - \phi(\mathbf{x}_{ij}) \right) \tag{3}$$

The values $\alpha_{i,j}$ are a set of "dual" parameters. It follows that the score of a candidate can be calculated using the dual parameters, and inner products between vectors, without having to explicitly deal with feature or parameter vectors in the space $\mathbb{R}^n$:

$\langle \bar{w}^*, \mathbf{x} \rangle =$
$\sum_{(i,j)} \alpha_{i,j} \left(\langle \phi(\mathbf{x}_{i1}), \phi(\mathbf{x}) \rangle - \langle \phi(\mathbf{x}_{ij}), \phi(\mathbf{x}) \rangle \right)$

Moreover, it turns out that many quadratic programming algorithms (e.g., see (Platt 1998)) can be applied using inner products between training examples alone, and without explicit representation in the feature space $\mathbb{R}^n$. Thus if the inner product between examples can be computed efficiently, SVMs can be trained and applied to the representation ϕ. This is important because for many representations $\langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle$ can be computed efficiently in spite of $\phi(\mathbf{x})$ being huge or even having infinite dimensions. This has been used to great effect in the classification literature to learn hyperplanes in spaces that would be intractable for other methods. See the papers in (Scholkopf et al. 1998) for many examples.

Figure 1 gives training and decoding methods for the SVM approach. The algorithms require a training set, an algorithm for enumerating candidate parses for each sentence, and an algorithm (kernel K) for computing $K(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle$ for some feature vector representation ϕ of examples. Under these assumptions the SVM can be trained and applied in the space defined by ϕ, with computational complexity depending on the cost of computing $K(\mathbf{x}, \mathbf{y})$ rather than the size of the feature space.

The Training Algorithm

Inputs: Training examples $\{s_i, t_i\}$ for $i = 1 \ldots m$. An algorithm that enumerates a candidate set $\mathcal{C}(s_i) = \{\mathbf{x}_{i,1} \ldots \mathbf{x}_{i,n_i}\}$ for each s_i. An algorithm (kernel) that computes the inner product $\langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle$ between feature vectors $\phi(\mathbf{x})$ and $\phi(\mathbf{y})$ representing trees $\mathbf{x}$ and $\mathbf{y}$. A constant C.

Output: Dual parameters $\alpha_{i,j}$ for $i = 1 \ldots m, j = 2 \ldots n_i$ which through Eq. 3 define the maximum margin hyperplane $\bar{w}^*$ in the feature space defined by ϕ.

Algorithm:

• For each training example s_i enumerate candidates $\mathcal{C}(s_i)$. W.l.g. take $\mathbf{x}_{i,1}$ to be t_i.

• Use a quadratic programming algorithm for SVMs (e.g., see (Platt 1998)) to find the dual parameters $\alpha_{i,j}$ which define the hyperplane that minimizes Eq. 1 subject to the constraints Eq. 2.

The Decoding Algorithm

Input: A set of candidates $\mathcal{C}(s)$ for an input sentence s. A set of examples $\{\mathbf{x}_{i,1} \ldots \mathbf{x}_{i,n_i}\}$ together with dual parameter values $\alpha_{i,j}$. An algorithm (kernel) that computes the inner product $\langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle$ between feature vectors.

Output: The highest scoring candidate under the hyperplane defined by the $\alpha_{i,j}$'s.

Algorithm: Return

$$\arg \max_{\mathbf{x} \in \mathcal{C}(s)} \sum_{i,j} \alpha_{i,j} \left(\langle \phi(\mathbf{x}_{i,1}), \phi(\mathbf{x}) \rangle - \langle \phi(\mathbf{x}_{i,j}), \phi(\mathbf{x}) \rangle \right)$$

Figure 1: Training and Decoding Algorithms for SVMs for NLP Problems.

3 Kernels for NLP Problems

3.1 A Kernel for Trees

We will consider learning algorithms that represent each tree x as a vector $\phi(x) = \{h_1(x), h_2(x), \ldots\}$. Each element $h_s(x)$ is a "feature" or function of the tree. The representation we will consider is a feature vector that counts all *sub-trees* seen in training data – this representation has previously been used by (Bod 1998). A sub-tree is a larger tree fragment that is formed by one or more applications of the rules in the grammar. See figure 2 for examples. This is a very high dimensional representation, a given tree having an exponential number of sub-trees.

We now show that a kernel can be used with this representation. We assume that each of the m unique subtrees in training data is indexed by an integer between 1 and m; we define $h_s(T)$ to be the number of times the s'th sub-tree in training data appears in T. The inner product between two trees under the representation $\phi(T) = \{h_1(T), h_2(T) \ldots h_m(T)\}$ is

$$K(T_1, T_2) = \langle \phi(T_1), \phi(T_2) \rangle = \sum_s h_s(T_1) h_s(T_2)$$

Computing this inner product directly is intractable: the sum is over an exponential number of sub-trees. To compute K efficiently we first define the set of nodes in trees T_1 and T_2 as N_1 and N_2 respectively. We define the indicator function $I_s(n)$ to be 1 if sub-tree s is seen rooted at node n, 0 otherwise. Hence

$$h_s(T_1) = \sum_{n_1 \in N_1} I_s(n_1) \quad h_s(T_2) = \sum_{n_2 \in N_2} I_s(n_2) \tag{4}$$

The first step to efficient computation of the inner product is the property

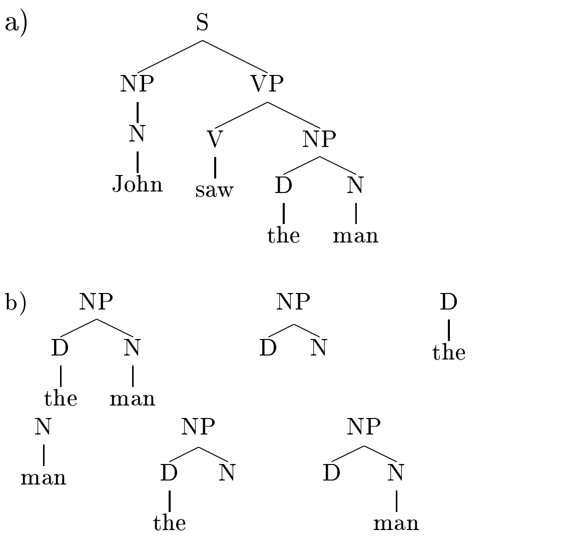

Figure 2: a) An example parse tree. b) The sub-trees of the NP covering *the man*. The tree in (a) contains all of these subtrees (some multiple times), as well as many others.

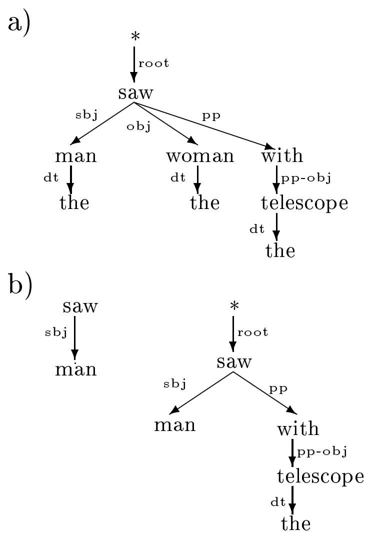

Figure 3: a) A dependency graph. An arrow signifies a dependency from a head-word to a modifier. The arrows are labeled with the grammatical relationship involved. b) Two sub-graphs of the dependency graph.

$$\begin{aligned}\langle \phi(T_1), \phi(T_2) \rangle &= \textstyle\sum_s h_s(T_1) h_s(T_2) \\ &= \textstyle\sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \sum_s I_s(n_1) I_s(n_2)\end{aligned}$$

This can be shown through application of the identities in (4). If we define $Cm(n_1, n_2)$ as

$$Cm(n_1, n_2) = \sum_s I_s(n_1) I_s(n_2)$$

it follows that

$$\langle \phi(T_1), \phi(T_2) \rangle \quad = \quad \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} Cm(n_1, n_2) \tag{5}$$

$Cm(n_1, n_2)$ is the number of common sub-trees rooted at both n_1 and n_2. It can be computed through a recursive definition:

- If productions at n_1 and n_2 are different then $Cm(n_1, n_2) = 0$.
- Else if productions at n_1/n_2 are the same, and both n_1/n_2 are pre-terminals, then $Cm(n_1, n_2) = 1$.
- Else if productions at n_1/n_2 are the same, and both n_1/n_2 are not pre-terminals.

$$Cm(n_1, n_2) = \prod_{i=1}^{nc(n_1)} (1 + Cm(ch(n_1, i), ch(n_2, i)))$$

In this last definition, $nc(n_1)$ is the number of non-terminals directly below n_1 in the tree; due to the productions at n_1 and n_2 being the same, we have $nc(n_1) = nc(n_2)$. $ch(n_1, i)$ is the i'th child-node of n_1.

The last 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 $(1 + Cm(ch(n_1, i), ch(n_2, i)))$ possible choices at the i'th child.

From equation (5), and the recursive definition of $Cm(n_1, n_2)$, $\langle \phi(T_1), \phi(T_2) \rangle$ can be calculated in $O(|N_1||N_2|)$ time. For our application this is a pessimistic estimate of the runtime: a more useful characterization is that it runs in linear time 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 a typically linear number of nodes with identical productions, so that running time is close to linear in the size of the trees.

3.2 A Kernel for Dependency Structures

This section describes a kernel for dependency structures. Figure 3(a) shows an example dependency structure. Each node in the tree is a word in the underlying sentence. Dependency relationships between pairs of words are shown as arrows from the head to its modifier; the relations are marked with labels such as "sbj" or "obj". At the root of the tree is a special start node, labeled "*". We assume that each head has at most one modifier with each possible relation.

We represent the graph as a set of nodes $\{w_1, \ldots, w_n\}$. The root node is w_1; the other nodes are ordered arbitrarily. For each node w, $word(w)$ is the word at the node. $children(w)$ is used to denote the set of $(relation, node)$ pairs which are direct modifiers of w. For example, the graph in figure 3(a) would have the properties

$word(w_1) =$ "*"
$children(w_1) = \{(root, w_2)\}$
$word(w_2) =$ "saw"
$children(w_2) = \{(sbj, w_3), (obj, w_4), (pp, w_5)\}$
$word(w_3) =$ "man"
...

A feature of a dependency graph is a connected sub-graph with at least two words. Figure 3(b) shows two example features. We will represent a dependency graph d as a vector of counts $\phi(d) = \{h_1(d), h_2(d), \ldots, h_m(d)\}$ where each $h_s(d)$ is the number of times a particular feature is seen within d. In a similar way to the tree kernel, the vector tracks all possible subgraphs. By very similar arguments to those in section 3.1, the inner product between two vectors can be calculated as

$$K(d_1, d_2) = \sum_{n_1 \in N_1, n_2 \in N_2} Cm(n_1, n_2)$$

where N_1 and N_2 are the sets of nodes in d_1 and d_2 respectively, and $Cm(n_1, n_2)$ is the number of common subgraphs rooted at n_1 and n_2. The definition for $Cm(n_1, n_2)$ is also similar. First we define $sim(n_1, n_2)$ as the set of common dependencies of n_1 and n_2:

$$\begin{aligned} sim(n_1, n_2) = \\ \{(x, y) \mid\ & (Relation, x) \in children(n_1), \\ & (Relation, y) \in children(n_2), \\ & word(x) = word(y)\} \end{aligned}$$

We can calculate Cm as follows:

- If $word(n_1) \neq word(n_2)$ or $children(n_1) = \emptyset$ or $children(n_2) = \emptyset$ then $Cm(n_1, n_2) = 0$
- Else
  $Cm(n_1, n_2) =$
  $\prod_{(x,y) \in sim(n_1, n_2)} (Cm(x, y) + 2) - 1$

The last part of the definition follows because a subtree rooted at n_1 and n_2 can be formed by making 1 of $(Cm(x, y) + 2)$ at choices at each shared node (x, y): either to pick 1 of the $Cm(x, y)$ sub-trees common to nodes x and y, or to just pick the single node at x/y, or to exclude the node altogether. This method overcounts by including the sub-tree formed by excluding all shared nodes, thereby forming a node which is n_1/n_2 alone: for this reason 1 is subtracted from the product.

3.3 A Kernel for Paired Sequences

We next describe a kernel for paired sequences, such as a correspondance between a sentence and its sequence of part-of-speech tags. Each tag is referred to as a "state", and each state underlies (or generates) a single word. An example of a paired sequence is `{* a/A b/C j/B d/E d/D j/H *}`, where states are in capital letters, "words" are in lower case. There are distinguished start and end states, labeled "*", which do not generate a word. For each node w, $label(w)$ is the word at the node. We define $next(w)$ to be the node in the sequence after w, and $word(w)$ to be the word generated by the state.

A sub-structure of a paired sequence consists of a sub-sequence of states, where each state may or may not have the word that it generated below it. In the usual way we represent an object p by a vector $\phi(p) = \{h_1(p), h_2(p), \ldots\}$ where $h_s(p)$ is the number of times the s'th sub-sequence is seen in p. The kernel is $\langle \phi(p_1), \phi_s(p_2) \rangle = \sum_{n_1 \in N_1, n_2 \in N_2} Cm(n_1, n_2)$ where N_1, N_2 are the sets of states in p_1, p_2 respectively, and $Cm(n_1, n_2)$ is the number of common sub-sequences beginning at nodes n_1, n_2 in p_1, p_2. There is again a recursive definition of Cm:

- If $label(n_1) \neq label(n_2)$ then $Cm(n_1, n_2) = 0$
- Else if $word(n_1) \neq word(n_2)$
  $Cm(n_1, n_2) = Cm(next(n_1), next(n_2))$

- Else
  $Cm(n_1, n_2) =$
  $1 + 2 * Cm(next(n_1), next(n_2))$

3.4 Downweighting Larger Structures

In practice, the kernels described in previous sections may weight larger substructures too highly. As evidence for this, we have found in experiments on WSJ treebank trees that the tree kernel between an example and itself is typically of order 10^6 while the average kernel value between different trees is of order 10^2. In this section we describe a modification to the tree kernel that addresses this problem; similar solutions can be applied to the dependency or paired-sequence kernels.

The reason that the tree kernel is so peaked is that there is an exponential blow-up in the number of sub-trees with their depth. The feature vector can be modified to downweight larger sub-trees: the new representation is

$$h_s(T_1) = \lambda^{size(s)} \sum_{n_1 \in N_1} I_s(n_1)$$

where $0 < \lambda \leq 1$ is a weighting parameter, and $size(s)$ is the number of rules within the sub-tree s. Thus components of ϕ which correspond to larger sub-trees are given lower weight. The inner product between vectors of this form can be computed in the way described in section 3.1, with the last two definitions for Cm being altered as follows:

- If productions at n_1/n_2 are the same, and n_1/n_2 are pre-terminals, $Cm(n_1, n_2) = \lambda^2$.
- Else if productions at n_1/n_2 are the same, and both n_1/n_2 are not pre-terminals.

$$Cm(n_1, n_2) = \lambda^2 \prod_{i=1}^{nc(n_1)} (1 + Cm(ch(n_1, i), ch(n_2, i)))$$

More generally, a different parameter $\lambda(x)$ could be associated with each context-free rule x, by replacing λ with $\lambda(rule-at(n_1))$ in the above definitions for Cm. This would result in the components (sub-trees) of ϕ being weighted by the product of λ's corresponding to the rules they contain. One choice would be to use the maximum likelihood estimate of a PCFG to define these parameters ($\lambda(\alpha \rightarrow \beta) = Count(\alpha \rightarrow \beta)/Count(\alpha)$). This would give larger weight to subtrees which are estimated to be more frequent by the PCFG.

	Perceptron	PCFG	Random	Best
P	0.75	0.72	0.59	0.91
R	0.74	0.67	0.60	0.88

Table 1: Results on the ATIS Corpus. P = labeled precision, R=labeled recall.

4 Experiments

As a proof of concept we applied the tree kernels to the Penn treebank ATIS corpus (Marcus et al. 1993). We divided the corpus into a training set of 879 sentences, and a test set of 302 sentences. A PCFG was trained on the training set, and a beam search was used to give a set of parses, with PCFG probabilities, for each of the training and test sentences. The experiments were performed using a variant of the voted perceptron algorithm (Freund & Schapire 1999). The voted perceptron is a relative of the SVM which can be used with kernels in the same way and is somewhat more computationally efficient. The voted perceptron examines one example at a time and evaluates its prediction $\mathrm{sgn}(\langle \bar{w}, \phi(\mathbf{x}_{i1}) - \phi(\mathbf{x}_{ij}) \rangle)$. If this prediction is incorrect then the example $\phi(\mathbf{x}_{i1}) - \phi(\mathbf{x}_{ij})$ is added to the current parameter vector $\bar{w}$. In these experiments we ran the perceptron through the data only once, we used subtrees of depth at most 2 and trained on at most 20 candidate trees for each sentence. During testing the algorithm had to choose the best parse tree from the top 100 candidates yielded by the PCFG, for each of 302 sentences. We compared the precision and recall of the perceptron using our tree kernel to the precision and recall of the highest probability parse from the PCFG, a randomly chosen parse from the candidates and the best parse from the candidates. The results are given in Table 1. Clearly, these results demonstrate that the use of linear models with carefully designed kernels is a promising approach. Considerable work remains to be done to extract their full potential, however.

5 Discussion

5.1 Previous Work

There is much previous work on the use of kernels when examples are vectors in some space $\mathbb{R}^n$. However, the space $\mathbb{R}^n$ is quite different from the discrete objects described in this paper, which motivated our work on new kernels. Kernels for discrete, recursive structures were addressed in a general way by (Haussler 1999) when he defined "Convolution Kernels" which involve a recursive calculation over the "parts" of a discrete structure. (Lodhi et al. 2001) describe convolution kernels for strings, the application being text categorization.

The parse tree representation in section 3.1 has been used in previous work by (Bod 1998). There are several key differences between his approach and ours though. The parameter estimation criterion in (Bod 1998) is quite different; the model has a different form (the score for a parse can not be expressed as an inner product between vectors). The score for a parse in the model of (Bod 1998) requires explicitly summing over all sub-trees of that tree; in practice Monte Carlo techniques are required to approximate this score, rather than calculating the scores exactly.

5.2 Other Learning Algorithms

There are methods other than Support Vector Machines that can be used with kernels to learn hyperplanes. The perceptron algorithm, which is guaranteed to find a separating hyperplane if it exists, was the original algorithm used with kernels (Aizerman et al. 1964). The voted perceptron (Freund & Schapire 1999) has been shown to be competitive with SVM methods, while being computationally less intensive. (Singer 2000) describes the use of Leveraged Vector Machines.

6 Conclusions

We have shown how Support Vector Machines can be applied to natural language problems, allowing computationally efficient learning in high dimensional feature spaces. In future we plan to do more detailed experiments, applying the methods to problems such as parsing or named entity recognition.

References

Abney, S. (1997). Stochastic attribute-value grammars. *Computational Linguistics, 23*, 597-618.

Abney, S., Schapire, R., & Singer, Y. (1999). Boosting Applied to Tagging and PP Attachment. In *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora.*

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.

Bartlett, P. and Shawe–Taylor, J. (1998). Generalization Performance of Support Vector Machines and Other Pattern Classifiers. In (Scholkopf et al. 1998).

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

Burges, C. (1998). A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):1-47.

Charniak, E. (1999). *A maximum-entropy-inspired parser* (Technical Report CS99-12). Department of Computer Science, Brown University, RI.

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

Cortes, C. & Vapnik, V. (1995). Support–Vector Networks. In *Machine Learning*, 20(3):273–297.

Della Pietra, S., Della Pietra, V., & Lafferty, J. (1997). Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 19*, 380–393.

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.

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

Joachims, T. (1998). Text Categorization with Support Vector Machines. In *Proceedings of the European Conference on Machine Learning (ECML).*

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*.

Kimeldorf, G., & Wahba, G. (1971). Some Results on Tchebycheffian Spline Functions. *Journal of Mathematics Analysis and Applications*, 33(1):82-95.

Lodhi, H., Christianini, N., Shawe-Taylor, J., & Watkins, C. (2001). Text Classification using String Kernels. To appear 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.

Platt, J. (1998). Fast Training of Support Vector Machines using Sequential Minimal Optimization. In (Scholkopf et al. 1998).

Scholkopf, B., Burges, C., & Smola, A. (eds.). (1998). *Advances in Kernel Methods – Support Vector Learning*, MIT Press.

Singer, Y. (2000). Leveraged Vector Machines. In Solla, S.A., Leen, T.K., & Muller, K.R., editors, *Advances in Neural Information Processing Systems 12*, pages 610–616, MIT Press.