Dependency Tree Kernels for Relation Extraction

Aron Culotta
University of Massachusetts
Amherst, MA 01002
USA
culotta@cs.umass.edu

Jeffrey Sorensen
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
USA
sorenj@us.ibm.com

Abstract
We extend previous work on tree kernels to estimate the similarity between the dependency trees of sentences. Using this kernel within a Support Vector Machine, we detect and classify relations between entities in the Automatic Content Extraction (ACE) corpus of news articles. We examine the utility of different features such as Wordnet hypernyms, parts of speech, and entity types, and find that the dependency tree kernel achieves a 20% F1 improvement over a “bag-of-words” kernel.

1 Introduction
The ability to detect complex patterns in data is limited by the complexity of the data’s representation. In the case of text, a more structured data source (e.g. a relational database) allows richer queries than does an unstructured data source (e.g. a collection of news articles). For example, current web search engines would not perform well on the query, “list all California-based CEOs who have social ties with a United States Senator.” Only a structured representation of the data can effectively provide such a list.

The goal of Information Extraction (IE) is to discover relevant segments of information in a data stream that will be useful for structuring the data. In the case of text, this usually amounts to finding mentions of interesting entities and the relations that join them, transforming a large corpus of unstructured text into a relational database with entries such as those in Table 1.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>Organization</td>
<td>Cupertino, CA</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Organization</td>
<td>Redmond, WA</td>
</tr>
</tbody>
</table>

Table 1: An example of extracted fields

We describe a relation extraction technique based on kernel methods. Kernel methods are non-parametric density estimation techniques that compute a kernel function between data instances, where a kernel function can be thought of as a similarity measure. Given a set of labeled instances, kernel methods determine the label of a novel instance by comparing it to the labeled training instances using this kernel function. Nearest neighbor classification and support-vector machines (SVMs) are two popular examples of kernel methods (Fukunaga, 1990; Cortes and Vapnik, 1995).

An advantage of kernel methods is that they can search a feature space much larger than could be represented by a feature extraction-based approach. This is possible because the kernel function can explore an implicit feature space when calculating the similarity between two instances, as described in the Section 3.

Working in such a large feature space can lead to over-fitting in many machine learning algorithms. To address this problem, we apply SVMs to the task of relation extraction. SVMs find a boundary between instances of different classes such that the distance between the boundary and the nearest instances is maximized. This characteristic, in addition to empirical validation, indicates that SVMs are particularly robust to over-fitting.

Here we are interested in detecting and classifying instances of relations, where a relation is some meaningful connection between two entities (Table 2). We represent each relation instance as an augmented dependency tree. A dependency tree represents the grammatical dependencies in a sentence; we augment this tree with features for each node...
(e.g. part of speech) We choose this representation because we hypothesize that instances containing similar relations will share similar substructures in their dependency trees. The task of the kernel function is to find these similarities.

We define a tree kernel over dependency trees and incorporate this kernel within an SVM to extract relations from newswire documents. The tree kernel approach consistently outperforms the bag-of-words kernel, suggesting that this highly-structured representation of sentences is more informative for detecting and distinguishing relations.

## 2 Related Work

Kernel methods (Vapnik, 1998; Cristianini and Shawe-Taylor, 2000) have become increasingly popular because of their ability to map arbitrary objects to a Euclidian feature space. Haussler (1999) describes a framework for calculating kernels over discrete structures such as strings and trees. String kernels for text classification are explored in Lodhi et al. (2000), and tree kernel variants are described in (Zelenko et al., 2003; Collins and Duffy, 2002; Cumby and Roth, 2003). Our algorithm is similar to that described by Zelenko et al. (2003). Our contributions are a richer sentence representation, a more general framework to allow feature weighting, as well as the use of composite kernels to reduce kernel sparsity.

Brin (1998) and Agichtein and Gravano (2000) apply pattern matching and wrapper techniques for relation extraction, but these approaches do not scale well to fastly evolving corpora. Miller et al. (2000) propose an integrated statistical parsing technique that augments parse trees with semantic labels denoting entity and relation types. Whereas Miller et al. (2000) use a generative model to produce parse information as well as relation information, we hypothesize that a technique discriminatively trained to classify relations will achieve better performance. Also, Roth and Yih (2002) learn a Bayesian network to tag entities and their relations simultaneously. We experiment with a more challenging set of relation types and a larger corpus.

## 3 Kernel Methods

In traditional machine learning, we are provided a set of training instances $S = \{x_1 \ldots x_N\}$, where each instance $x_i$ is represented by some $d$-dimensional feature vector. Much time is spent on the task of feature engineering — searching for the optimal feature set either manually by consulting domain experts or automatically through feature induction and selection (Scott and Matwin, 1999). For example, in entity detection the original instance representation is generally a word vector corresponding to a sentence. Feature extraction and induction may result in features such as part-of-speech, word n-grams, character n-grams, capitalization, and conjunctions of these features. In the case of more structured objects, such as parse trees, features may include some description of the object’s structure, such as “has an NP-VP subtree.” Kernel methods can be particularly effective at reducing the feature engineering burden for structured objects. By calculating the similarity between two objects, kernel methods can employ dynamic programming solutions to efficiently enumerate over substructures that would be too costly to explicitly include as features.

Formally, a kernel function $K$ is a mapping $K : \mathbb{X} \times \mathbb{X} \rightarrow [0, \infty]$ from instance space $\mathbb{X}$ to a similarity score $K(x, y) = \sum_i \phi_i(x) \phi_i(y) = \phi(x) \cdot \phi(y)$. Here, $\phi_i(x)$ is some feature function over the instance $x$. The kernel function must be symmetric $[K(x, y) = K(y, x)]$ and positive-semidefinite. By positive-semidefinite, we require that if $x_1, \ldots, x_n \in \mathbb{X}$, then the $n \times n$ matrix $G$ defined by $G_{ij} = K(x_i, x_j)$ is positive semi-definite. It has been shown that any function that takes the dot product of feature vectors is a kernel function (Haussler, 1999).

A simple kernel function takes the dot product of the vector representation of instances being compared. For example, in document classification, each document can be represented by a binary vector, where each element corresponds to the presence or absence of a particular word in that document. Here, $\phi_i(x) = 1$ if word $i$ occurs in document $x$. Thus, the kernel function $K(x, y)$ returns the num-

<table>
<thead>
<tr>
<th>AT</th>
<th>NEAR</th>
<th>PART</th>
<th>ROLE</th>
<th>SOCIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based-In</td>
<td>Relative-location</td>
<td>Part-of Subsidiary</td>
<td>Affiliate, Founder</td>
<td>Associate, Grandparent</td>
</tr>
<tr>
<td>Located</td>
<td></td>
<td>Other</td>
<td>Citizen-of, Management</td>
<td>Parent, Sibling</td>
</tr>
<tr>
<td>Residence</td>
<td></td>
<td></td>
<td>Client, Member</td>
<td>Spouse, Other-professional</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Owner, Other, Staff</td>
<td>Other-relative, Other-personal</td>
</tr>
</tbody>
</table>
ber of words in common between $x$ and $y$. We refer to this kernel as the “bag-of-words” kernel, since it ignores word order.

When instances are more structured, as in the case of dependency trees, more complex kernels become necessary. Haussler (1999) describes convolution kernels, which find the similarity between two structures by summing the similarity of their substructures. As an example, consider a kernel over strings. To determine the similarity between two strings, string kernels (Lodhi et al., 2000) count the number of common subsequences in the two strings, and weight these matches by their length. Thus, $\phi_i(x)$ is the number of times string $x$ contains the subsequence referenced by $i$. These matches can be found efficiently through a dynamic program, allowing string kernels to examine long-range features that would be computationally infeasible in a feature-based method.

Given a training set $S = \{x_1, \ldots, x_N\}$, kernel methods compute the Gram matrix $G$ such that $G_{ij} = K(x_i, x_j)$. Given $G$, the classifier finds a hyperplane which separates instances of different classes. To classify an unseen instance $x$, the classifier first projects $x$ into the feature space defined by the kernel function. Classification then consists of determining on which side of the separating hyperplane $x$ lies.

A support vector machine (SVM) is a type of classifier that formulates the task of finding the separating hyperplane as the solution to a quadratic programming problem (Cristianini and Shawe-Taylor, 2000). Support vector machines attempt to find a hyperplane that not only separates the classes but also maximizes the margin between them. The hope is that this will lead to better generalization performance on unseen instances.

4 Augmented Dependency Trees

Our task is to detect and classify relations between entities in text. We assume that entity tagging has been performed; so to generate potential relation instances, we iterate over all pairs of entities occurring in the same sentence. For each entity pair, we create an augmented dependency tree (described below) representing this instance. Given a labeled training set of potential relations, we define a tree kernel over dependency trees which we then use in an SVM to classify test instances.

A dependency tree is a representation that denotes grammatical relations between words in a sentence (Figure 1). A set of rules maps a parse tree to a dependency tree. For example, subjects are dependent on their verbs and adjectives are dependent on the nouns they modify. Note that for the purposes of this paper, we do not consider the link labels (e.g. “object”, “subject”); instead we use only the dependency structure. To generate the parse tree of each sentence, we use MXPOST, a maximum entropy statistical parser$^1$; we then convert this parse tree to a dependency tree. Note that the left-to-right ordering of the sentence is maintained in the dependency tree only among siblings (i.e. the dependency tree does not specify an order to traverse the tree to recover the original sentence).

For each pair of entities in a sentence, we find the smallest common subtree in the dependency tree that includes both entities. We choose to use this subtree instead of the entire tree to reduce noise and emphasize the local characteristics of relations. We then augment each node of the tree with a feature vector (Table 3). The relation-argument feature specifies whether an entity is the first or second argument in a relation. This is required to learn asymmetric relations (e.g. $X$ owns $Y$).

Formally, a relation instance is a dependency tree

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Feature & Example \\
\hline
word & troops, Tikrit \\
part-of-speech (24 values) & NN, NNP \\
general-pos (5 values) & noun, verb, adj \\
chunk-tag & NP, VP, ADJP \\
entity-type & person, geo-political-entity \\
entity-level & name, nominal, pronoun \\
Wordnet hypernyms & social group, city \\
relation-argument & ARG_A, ARG_B \\
\hline
\end{tabular}
\caption{List of features assigned to each node in the dependency tree.}
\end{table}

$^1$http://www.cis.upenn.edu/~adwait/statnlp.html
with nodes \( \{t_0 \ldots t_n\} \). The features of node \( t_i \) are given by \( \phi(t_i) = \{v_1 \ldots v_d\} \). We refer to the \( j^{th} \) child of node \( t_i \) as \( t_i[j] \), and we denote the set of all children of node \( t_i \) as \( t_i[c] \). We reference a subset \( j \) of children of \( t_i \) by \( t_i[j] \subseteq t_i[c] \). Finally, we refer to the parent of node \( t_i \) as \( t_i.p \).

From the example in Figure 1, \( t_0[1] = t_2, t_0[\{0,1\}] = \{t_1, t_2\} \), and \( t_1.p = t_0 \).

5 Tree kernels for dependency trees

We now define a kernel function for dependency trees. The tree kernel is a function \( K(T_1, T_2) \) that returns a normalized, symmetric similarity score in the range \((0,1)\) for two trees \( T_1 \) and \( T_2 \). We define a slightly more general version of the kernel described by Zelenko et al. (2003).

We first define two functions over the features of tree nodes: a matching function \( m(t_i, t_j) \in \{0,1\} \) and a similarity function \( s(t_i, t_j) \in (0,\infty] \). Let the feature vector \( \phi(t_i) = \{v_1 \ldots v_d\} \) consist of two possibly overlapping subsets \( \phi_m(t_i) \subseteq \phi(t_i) \) and \( \phi_s(t_i) \subseteq \phi(t_i) \). We use \( \phi_m(t_i) \) in the matching function and \( \phi_s(t_i) \) in the similarity function. We define

\[
m(t_i, t_j) = \begin{cases} 
1 & \text{if } \phi_m(t_i) = \phi_m(t_j) \\
0 & \text{otherwise}
\end{cases}
\]

and

\[
s(t_i, t_j) = \sum_{v_q \in \phi_s(t_i)} \sum_{v_r \in \phi_s(t_j)} C(v_q, v_r)
\]

where \( C(v_q, v_r) \) is some compatibility function between two feature values. For example, in the simplest case where

\[
C(v_q, v_r) = \begin{cases} 
1 & \text{if } v_q = v_r \\
0 & \text{otherwise}
\end{cases}
\]

\( s(t_i, t_j) \) returns the number of feature values in common between feature vectors \( \phi_s(t_i) \) and \( \phi_s(t_j) \).

We can think of the distinction between functions \( m(t_i, t_j) \) and \( s(t_i, t_j) \) as a way to discretize the similarity between two nodes. If \( \phi_m(t_i) \neq \phi_m(t_j) \), then we declare the two nodes completely dissimilar. However, if \( \phi_m(t_i) = \phi_m(t_j) \), then we proceed to compute the similarity \( s(t_i, t_j) \). Thus, restricting nodes by \( m(t_i, t_j) \) is a way to prune the search space of matching subtrees, as shown below.

For two dependency trees \( T_1, T_2 \), with root nodes \( r_1 \) and \( r_2 \), we define the tree kernel \( K(T_1, T_2) \) as follows:

\[
K(T_1, T_2) = \begin{cases} 
0 & \text{if } m(r_1, r_2) = 0 \\
\lambda s(r_1, r_2) + K_c(r_1[c], r_2[c]) & \text{otherwise}
\end{cases}
\]

where \( K_c \) is a kernel function over children. Let \( \lambda \) and \( \rho \) be sequences of indices such that \( \lambda \) is a sequence \( \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n \), and likewise for \( \rho \). Let \( d(\lambda) = \lambda_n - \lambda_1 + 1 \) and \( l(\lambda) \) be the length of \( \lambda \). Then we have \( K_c(t_i[c], t_j[c]) = \sum_{a,b,l(a)=l(b)} \chi^d(a) \chi^d(b) K(t_i[a], t_j[b]) \)

The constant \( 0 < \lambda < 1 \) is a decay factor that penalizes matching subsequences that are spread out within the child sequences. See Zelenko et al. (2003) for a proof that \( K \) is kernel function.

Intuitively, whenever we find a pair of matching nodes, we search for all matching subsequences of the children of each node. A matching subsequence of children is a sequence of children \( \lambda \) and \( \rho \) such that \( m(\lambda_i, \rho_i) = 1 \) (\( \forall i < n \)). For each matching pair of nodes \( (\lambda_i, \rho_i) \) in a matching subsequence, we accumulate the result of the similarity function \( s(\lambda_i, \rho_i) \) and then recursively search for matching subsequences of their children \( \lambda_i[c], \rho_i[c] \).

We implement two types of tree kernels. A contiguous kernel only matches children subsequences that are uninterrupted by non-matching nodes. Therefore, \( d(\lambda) = l(\lambda) \). A sparse tree kernel, by contrast, allows non-matching nodes within matching subsequences.

Figure 2 shows two relation instances, where each node contains the original text plus the features used for the matching function, \( \phi_m(t_i) = \{\text{general}-\text{pos}, \text{entity}-\text{type}, \text{relation}-\text{argument}\} \). (“NA” denotes the feature is not present for this node.) The contiguous kernel matches the following substructures: \( \{t_0[0], u_0[0]\}, \{t_0[2], u_0[1]\}, \{t_3[0], u_2[0]\} \). Because the sparse kernel allows non-contiguous matching sequences, it matches an additional substructure \( \{t_0[0]*, 2], u_0[0, *, 1]\} \), where \( * \) indicates an arbitrary number of non-matching nodes.

Zelenko et al. (2003) have shown the contiguous kernel to be computable in \( O(mn) \) and the sparse kernel in \( O(mn^3) \), where \( m \) and \( n \) are the number of children in trees \( T_1 \) and \( T_2 \) respectively.

6 Experiments

We extract relations from the Automatic Content Extraction (ACE) corpus provided by the National Institute for Standards and Technology (NIST). The
Figure 2: Two instances of the NEAR relation.

data consists of about 800 annotated text documents gathered from various newspapers and broadcasts. Five entities have been annotated (PERSON, ORGANIZATION, GEO-POLITICAL ENTITY, LOCATION, FACILITY), along with 24 types of relations (Table 2). As noted from the distribution of relationship types in the training data (Figure 3), data imbalance and sparsity are potential problems.

In addition to the contiguous and sparse tree kernels, we also implement a bag-of-words kernel, which treats the tree as a vector of features over nodes, disregarding any structural information. We also create composite kernels by combining the sparse and contiguous kernels with the bag-of-words kernel. Joachims et al. (2001) have shown that given two kernels \( K_1, K_2 \), the composite kernel \( K_{12}(x_i, x_j) = K_1(x_i, x_j) + K_2(x_i, x_j) \) is also a kernel. We find that this composite kernel improves performance when the Gram matrix \( G \) is sparse (i.e. our instances are far apart in the kernel space).

The features used to represent each node are shown in Table 3. After initial experimentation, the set of features we use in the matching function is \( \phi_m(t_i) = \{ \text{general-pos, entity-type, relation-argument} \} \), and the similarity function examines the remaining features.

In our experiments we tested the following five kernels:

\[
\begin{align*}
K_0 &= \text{sparse kernel} \\
K_1 &= \text{contiguous kernel} \\
K_2 &= \text{bag-of-words kernel} \\
K_3 &= K_0 + K_2 \\
K_4 &= K_1 + K_2
\end{align*}
\]

We also experimented with the function \( C(v_q, v_r) \), the compatibility function between two feature values. For example, we can increase the importance of two nodes having the same Wordnet hypernym. If \( v_q, v_r \) are hypernym features, then we can define

\[
C(v_q, v_r) = \begin{cases} 
\alpha & \text{if } v_q = v_r \\
0 & \text{otherwise}
\end{cases}
\]

When \( \alpha > 1 \), we increase the similarity of nodes that share a hypernym. We tested a number of weighting schemes, but did not obtain a set of weights that produced consistent significant improvements. See Section 8 for alternate approaches to setting \( C \).

---

3http://www.cogsci.princeton.edu/~wn/
Table 4: Kernel performance comparison.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_1$</td>
<td>69.6</td>
<td>25.3</td>
<td>36.8</td>
</tr>
<tr>
<td>$K_2$</td>
<td>47.0</td>
<td>10.0</td>
<td>14.2</td>
</tr>
<tr>
<td>$K_3$</td>
<td>68.9</td>
<td>24.3</td>
<td>35.5</td>
</tr>
<tr>
<td>$K_4$</td>
<td><strong>70.3</strong></td>
<td><strong>26.3</strong></td>
<td><strong>38.0</strong></td>
</tr>
</tbody>
</table>

Table 5: Relation detection performance. (B) denotes binary classification.

<table>
<thead>
<tr>
<th></th>
<th>Prec.</th>
<th>Rec.</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_0$ (B)</td>
<td>83.4</td>
<td>45.5</td>
<td>58.8</td>
</tr>
<tr>
<td>$K_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_1$ (B)</td>
<td>91.4</td>
<td>37.1</td>
<td>52.8</td>
</tr>
<tr>
<td>$K_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_2$ (B)</td>
<td>92.7</td>
<td>10.6</td>
<td>19.0</td>
</tr>
<tr>
<td>$K_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_3$ (B)</td>
<td>91.3</td>
<td>35.1</td>
<td>50.8</td>
</tr>
<tr>
<td>$K_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_4$ (B)</td>
<td>91.8</td>
<td>37.5</td>
<td>53.3</td>
</tr>
</tbody>
</table>

Table 6: Results on the cascading classification. D and C denote the kernel used for relation detection and classification, respectively.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_0$</td>
<td>$K_0$</td>
<td>66.0</td>
<td>29.0</td>
<td>40.1</td>
</tr>
<tr>
<td>$K_1$</td>
<td>$K_1$</td>
<td>66.6</td>
<td>32.4</td>
<td>43.5</td>
</tr>
<tr>
<td>$K_2$</td>
<td>$K_2$</td>
<td>62.5</td>
<td>27.7</td>
<td>38.1</td>
</tr>
<tr>
<td>$K_3$</td>
<td>$K_3$</td>
<td><strong>67.5</strong></td>
<td>34.3</td>
<td>45.3</td>
</tr>
<tr>
<td>$K_4$</td>
<td>$K_4$</td>
<td>67.1</td>
<td><strong>35.0</strong></td>
<td><strong>45.8</strong></td>
</tr>
<tr>
<td>$K_1$</td>
<td>$K_4$</td>
<td>67.4</td>
<td>33.9</td>
<td>45.0</td>
</tr>
<tr>
<td>$K_4$</td>
<td>$K_1$</td>
<td>65.3</td>
<td>32.5</td>
<td>43.3</td>
</tr>
</tbody>
</table>

To remedy this, we retrain each SVMs for a binary classification task. Here, we detect, but do not classify, relations. This allows us to combine all positive relation instances into one class, which provides us more training samples to estimate the class boundary. We then threshold our output to achieve an optimal operating point. As seen in Table 5, this method of relation detection outperforms that of the multi-class classifier.

We then use these binary classifiers in a cascading scheme as follows: First, we use the binary SVM to detect possible relations. Then, we use the SVM trained only on positive relation instances to classify each predicted relation. These results are shown in Table 6.

The first result of interest is that the sparse tree kernel, $K_0$, does not perform as well as the contiguous tree kernel, $K_1$. Suspecting that noise was introduced by the non-matching nodes allowed in the sparse tree kernel, we performed the experiment with different values for the decay factor $\lambda = \{.9, .5, .1\}$, but obtained no improvement.

The second result of interest is that all tree kernels outperform the bag-of-words kernel, $K_2$, most noticeably in recall performance, implying that the structural information the tree kernel provides is extremely useful for relation detection.

Note that the average results reported here are representative of the performance per relation, except for the NEAR relation, which had slightly lower results overall due to its infrequency in training.

7 Conclusions

We have shown that using a dependency tree kernel for relation extraction provides a vast improvement over a bag-of-words kernel. While the dependency tree kernel appears to perform well at the task of classifying relations, recall is still relatively low. Detecting relations is a difficult task for a kernel method because the set of all non-relation instances is extremely heterogeneous, and is therefore difficult to characterize with a similarity metric. An improved system might use a different method to detect candidate relations and then use this kernel method to classify the relations.

8 Future Work

The most immediate extension is to automatically learn the feature compatibility function $C(v_q, v_r)$.
A first approach might use tf-idf to weight each feature. Another approach might be to calculate the information gain for each feature and use that as its weight. A more complex system might learn a weight for each pair of features; however this seems computationally infeasible for large numbers of features.

One could also perform latent semantic indexing to collapse feature values into similar “categories”—for example, the words “football” and “baseball” might fall into the same category. Here, \( C(v_q, v_r) \) might return \( \alpha_1 \) if \( v_q = v_r \), and \( \alpha_2 \) if \( v_q \) and \( v_r \) are in the same category, where \( \alpha_1 > \alpha_2 > 0 \). Any method that provides a “soft” match between feature values will sharpen the granularity of the kernel and enhance its modeling power.

Further investigation is also needed to understand why the sparse kernel performs worse than the contiguous kernel. These results contradict those given in Zelenko et al. (2003), where the sparse kernel achieves 2-3% better F1 performance than the contiguous kernel. It is worthwhile to characterize relation types that are better captured by the sparse kernel, and to determine when using the sparse kernel is worth the increased computational burden.

References


