A Dependency-based Word Subsequence Kernel

Rohit J. Kate
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712-0233, USA
rjkate@cs.utexas.edu

Abstract

This paper introduces a new kernel which computes similarity between two natural language sentences as the number of paths shared by their dependency trees. The paper gives a very efficient algorithm to compute it. This kernel is also an improvement over the word subsequence kernel because it only counts linguistically meaningful word subsequences which are based on word dependencies. It overcomes some of the difficulties encountered by syntactic tree kernels as well. Experimental results demonstrate the advantage of this kernel over word subsequence and syntactic tree kernels.

1 Introduction

Kernel-based learning methods (Vapnik, 1998) are becoming increasingly popular in natural language processing (NLP) because they allow one to work with potentially infinite number of features without explicitly constructing or manipulating them. In most NLP problems, the data is present in structured forms, like strings or trees, and this structural information can be effectively passed to a kernel-based learning algorithm using an appropriate kernel, like a string kernel (Lodhi et al., 2002) or a tree kernel (Collins and Duffy, 2001). In contrast, feature-based methods require reducing the data to a pre-defined set of features often leading to some loss of the useful structural information present in the data.

A kernel is a measure of similarity between every pair of examples in the data and a kernel-based machine learning algorithm accesses the data only through these kernel values. For example, the string kernel (Lodhi et al., 2002; Cancedda et al., 2003) computes the similarity between two natural language strings as the number of common word subsequences between them. A subsequence allows gaps between the common words which are penalized according to a parameter. Each word subsequence hence becomes an implicit feature used by the kernel-based machine learning algorithm. A problem with this kernel is that many of these word subsequences common between two strings may not be semantically expressive or linguistically meaningful[1]. Another problem with this kernel is that if there are long-range dependencies between the words in a common word subsequence, then they will unfairly get heavily penalized because of the presence of word gaps.

The syntactic tree kernel presented in (Collins and Duffy, 2001) captures the structural similarity between two syntactic trees as the number of syntactic subtrees common between them. However, often syntactic parse trees may share syntactic subtrees which correspond to very different semantics based on what words they represent in the sentence. On the other hand, some subtrees may differ syntactically but may represent similar underlying semantics. These differences can become particularly problematic if the tree kernel is to be used for tasks which require semantic processing.

This paper presents a new kernel which computes similarity between two sentences as the the number of paths common between their dependency trees.

[1](Lodhi et al., 2002) use character subsequences instead of word subsequences which are even less meaningful.

Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing, pages 400–409,
Honolulu, October 2008.

(a) *A fat cat was chased by a dog.*
(b) *A cat with a red collar was chased two days ago by a fat dog.*

Figure 1: Two natural language sentences.

It improves over the word subsequence kernel because it only counts the word subsequences which are linked by dependencies. It also circumvents some of the difficulties encountered with the syntactic tree kernel when applied for semantic processing tasks.

Although several dependency-tree-based kernels and modifications to syntactic tree kernels have been proposed which we briefly discuss in the Related Work section, to our best knowledge no previous work has presented a kernel based on dependency paths which offers some unique advantages. We also give a very efficient algorithm to compute this kernel. We present experimental results on the task of domain-specific semantic parsing demonstrating the advantage of this kernel over word subsequence and syntactic tree kernels.

The following section gives some background on string and tree kernels. Section 3 then introduces the dependency-based word subsequence kernel and gives an efficient algorithm to compute it. Some of the related work is discussed next, followed by experiments, future work and conclusions.

2 String and Tree Kernels

2.1 Word-Subsequence Kernel

A kernel between two sentences measures the similarity between them. Lodhi et al. (2002) presented a string kernel which measures the similarity between two sentences, or two documents in general, as the number of character subsequences shared between them. This was extended by Cancedda et al. (2003) to the number of common word subsequences between them. We will refer to this kernel as the *word subsequence kernel*.

Consider the two sentences shown in Figure 1. Some common word subsequences between them are "a cat", "was chased by", "by a dog", "a cat chased by a dog", etc. Note that the subsequence "was chased by" is present in the second sentence but it requires skipping the words "two days ago" or has a gap of three words present in it. The kernel downweights the presence of gaps by a decay factor $\lambda \epsilon (0, 1]$. If g_1 and g_2 are the sum totals of gaps for a subsequence present in the two sentences respectively, then the contribution of this subsequence towards the kernel value will be $\lambda^{g_1+g_2}$. The kernel can be normalized to have values in the range of $[0, 1]$ to remove any bias due to different sentence lengths. Lodhi et al. (2002) give a dynamic programming algorithm to compute string subsequence kernels in $O(nst)$ time where s and t are the lengths of the two input strings and n is the maximum length of common subsequences one wants to consider. Rousu and Shawe-Taylor (2005) present an improved algorithm which works faster when the vocabulary size is large. Subsequence kernels have been used with success in NLP for text classification (Lodhi et al., 2002; Cancedda et al., 2003), information extraction (Bunescu and Mooney, 2005b) and semantic parsing (Kate and Mooney, 2006).

There are, however, some shortcomings of this word subsequence kernel as a measure of similarity between two sentences. Firstly, since it considers all possible common subsequences, it is not sensitive to whether the subsequence is linguistically meaningful or not. For example, the meaningless subsequences "cat was by" and "a was a" will also be considered common between the two sentences by this kernel. Since these subsequences will be used as implicit features by the kernel-based machine learning algorithm, their presence can only hurt the performance. Secondly, if there are long distance dependencies between the words of the subsequence present in a sentence then the subsequence will get unfairly penalized. For example, the most important word subsequence shared between the two sentences shown in Figure 1 is "a cat was chased by a dog" which will get penalized by total gap of eight words coming from the second sentence and a gap of one word from the first sentence. Finally, the kernel is not sensitive to the relations between the words, for example, the kernel will consider "a fat dog" as a common subsequence although in the first sentence "a fat" relates to the cat and not to the dog.

2.2 Syntactic Tree Kernel

Syntactic tree kernels were first introduced by Collins and Duffy (2001) and were also used by

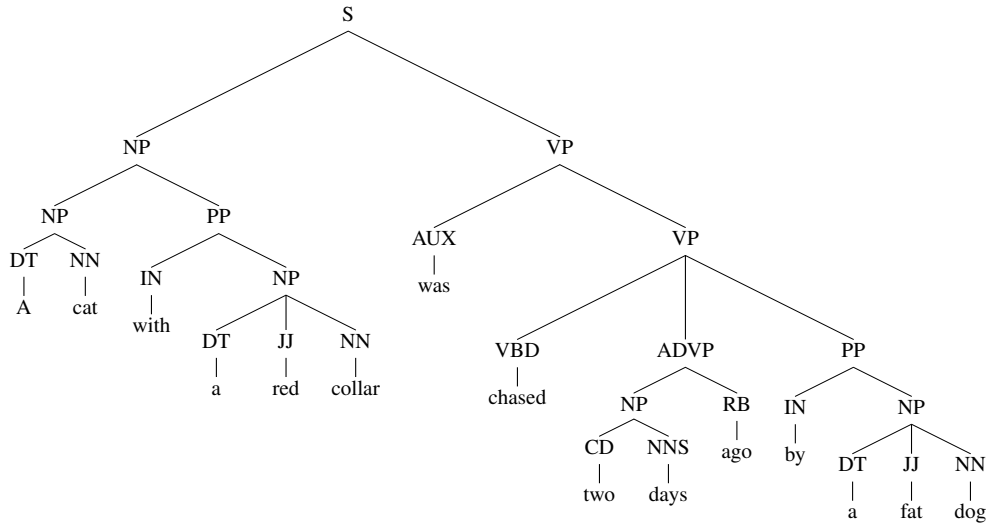

Figure 3: Syntactic parse tree of the sentence shown in Figure 1 (b).

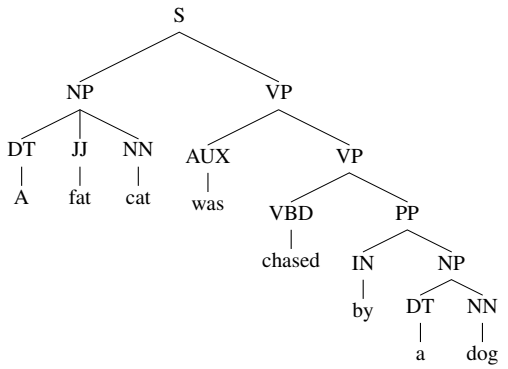

Figure 2: Syntactic parse tree of the sentence shown in Figure 1 (a).

Collins (2002) for the task of re-ranking syntactic parse trees. They define a kernel between two trees as the number of subtrees shared between them. A subtree is defined as any subgraph of the tree which includes more than one node, with the restriction that entire productions must be included at every node. The kernel defined this way captures most of the structural information present in the syntactic parse trees in the form of tree fragments which the kernelized learning algorithms can then implicitly use as features. The kernel can be computed in $O(|N_1||N_2|)$ time, where $|N_1|$ and $|N_2|$ are the number of nodes of the two trees. An efficient algorithm to compute tree kernels was given by Moschitti (2006a) which runs in close to linear time in the size of the input trees.

One drawback of this tree kernel, though, particularly when used for any task requiring semantic processing, is that it may match syntactic subtrees between two trees even though they represent very dissimilar things in the sentence. For example, between the syntactic parse trees shown in Figures 2 and 3 for the two sentences shown in Figure 1, the syntactic tree kernel will find (NP (DT a) JJ NN) as a common subtree but in the first sentence it represents "cat" while in the second it represents "collar" and "dog". It will also find "(NP (DT a) (JJ fat) NN)" as a common subtree which again refers to "cat" in the first sentence and "dog" in the second sentence. As another example, consider two simple sentences: (S (NP Chip) (VP (V saw) (NP Dale))) and (S (NP Mary) (VP (V heard) (NP Sally))). Even though semantically nothing is similar between them, the syntactic tree kernel will still find common subtrees (S NP VP), (VP N NP) and (S NP (VP V NP)). The underlying problem is that the syntactic tree kernel tends to overlook the words of the sentences which, in fact, carry the essential semantics. On the other hand, although (NP (DT a) (NN cat)) and (NP (DT a) (JJ fat) (NN cat)) represent very similar concepts but the kernel will not capture this high level similarity between the two constituents, and will only find (DT a) and (NN cat) as the common substructures. Finally, the most important similarity between the two sentences is "a cat was chased by a dog" which will not be captured by this kernel because

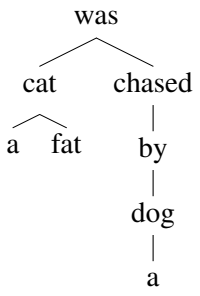

Figure 4: Dependency tree of the sentence shown in Figure 1 (a).

(b)

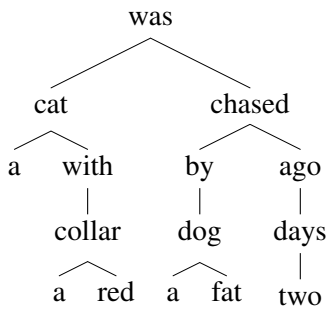

Figure 5: Dependency tree of the sentence shown in Figure 1 (b).

there is no common subtree which covers it. The Related Work section discusses some modifications that have been proposed to the syntactic tree kernel.

3 A Dependency-based Word Subsequence Kernel

A dependency tree encodes functional relationships between the words in a sentence (Hudson, 1984). The words of the sentence are the nodes and if a word complements or modifies another word then there is a child to parent edge from the first word to the second word. Every word in a dependency tree has exactly one parent except for the root word. Figures 4 and 5 show dependency trees for the two sentences shown in Figure 1. There has been a lot of progress in learning dependency tree parsers (McDonald et al., 2005; Koo et al., 2008; Wang et al., 2008). They can also be obtained indirectly from syntactic parse trees utilizing the head words of the constituents.

We introduce a new kernel which takes the words into account like the word-subsequence kernel and also takes the syntactic relations between them into account like the syntactic tree kernel, however, it does not have the shortcomings of the two kernels pointed out in the previous section. This kernel counts the number of common paths between the dependency trees of the two sentences. Another way to look at this kernel is that it counts all the common word subsequences which are linked by dependencies. Hence we will call it a *dependency-based word subsequence kernel*. Since the implicit features it uses are dependency paths which are enumerable, it is a well defined kernel. In other words, an example gets implicitly mapped to the feature space in which each dependency path is a dimension.

The dependency-based word subsequence kernel will find the common paths 'a → cat', 'cat → was ← chased', 'chased ← by ← dog' among many others between the dependency trees shown in Figures 4 and 5. The arrows are always shown from child node to the parent node. A common path takes into account the direction between the words as well. Also note that it will find the important subsequence 'a → cat → was ← chased ← by ← dog ← a' as a common path.

It can be seen that the word subsequences this kernel considers as common paths are linguistically meaningful. It is also not affected by long-range dependencies between words because those words are always directly linked in a dependency tree. There is no need to allow gaps in this kernel either because related words are always linked. It also won't find 'a fat' as a common path because in the first tree "cat" is between the two words and in the second sentence "dog" is between them. Thus it does not have the shortcomings of the word subsequence kernel. It also avoids the shortcomings of the syntactic tree kernel because the common paths are words themselves and syntactic labels do not interfere in capturing the similarity between the two sentences. It will not find anything common between dependency trees for the sentences "Chip saw Dale" and "Mary heard Sally". But it will find 'a → cat' as a common path between "a cat" and "a fat cat". We however note that this kernel does not use general syntactic categories, unlike the syntactic tree kernel, which will limit its applicability to the tasks which depend on the syntactic categories, like re-ranking syntactic parse trees.

We now give an efficient algorithm to compute all the common paths between two trees. To our best knowledge, no previous work has considered this problem. The key observation for this algorithm is that a path in a tree always has a structure in which nodes (possibly none) go up to a highest node followed by nodes (possibly none) coming down. Based on this observation we compute two quantities for every pair of nodes between the two trees. We call the first quantity *common downward paths* (CDP) between two nodes, one from each tree, and it counts the number of common paths between the two trees which originate from those two nodes and *which always go downward*. For example, the common downward paths between the 'chased' node of the tree in Figure 4 and the 'chased' node of the tree in Figure 5 are 'chased ← by', 'chased ← by ← dog' and 'chased ← by ← dog ← a'. Hence $CDP(chased, chased) = 3$. A word may occur multiple times in a sentence so the CDP values will be computed separately for each occurrence. We will shortly give a fast recursive algorithm to compute CDP values.

Once these CDP values are known, using these the second quantity is computed which we call *common peak paths* (CPP) between every two nodes, one from each tree. This counts the number of common paths between the two trees which *peak* at those two nodes, i.e. these nodes are the highest nodes in those paths. For example, 'was' is the peak for the path 'a → cat → was ← chased'. Since every common path between the two trees has a unique highest node, once these CPP values have been computed, the number of common paths between the two trees is simply the sum of all these CPP values.

We now describe how all these values are efficiently computed. The CDP values between every two nodes n_1 and n_2 of the trees T_1 and T_2 respectively, is recursively computed as follows:

$$CDP(n_1, n_2) = 0 \; if \; n_1.w \neq n_2.w$$

otherwise,

$$CDP(n_1, n_2) = \sum_{\substack{c_1 \epsilon C(n_1) \\ c_2 \epsilon C(n_2) \\ c_1.w = c_2.w}} (1 + CDP(c_1, c_2))$$

In the first equation, $n.w$ stands for the word at the node n. If the words are not equal then there cannot be any common downward paths originating from the nodes. In the second equation, $C(n)$ represents the set of children nodes of the node n in a tree. If the words at two children nodes are the same, then the number of common downward paths from the parent will include all the common downward paths at the two children nodes incremented with the link from the parent to the children. In addition the path from parent to the child node is also a common downward path. For example, in the trees shown in Figures 4 and 5, the nodes with word 'was' have 'chased' as a common child. Hence all the common downward paths originating from 'chased' (namely 'chased ← by', 'chased ← by ← dog' and 'chased ← by ← dog ← a') when incremented with 'was ← chased' become common downward paths originating from 'was'. In addition, the path 'was ← chased' itself is a common downward path. Since 'cat' is also a common child at 'was', it's common downward paths will also be added.

The CDP values thus computed are then used to compute the CPP values as follows:

$$CPP(n_1, n_2) = 0 \; if \; n_1.w \neq n_2.w$$

otherwise,

$$CPP(n_1, n_2) = CDP(n_1, n_2) + \sum_{\substack{c_1, \hat{c_1} \epsilon C(n_1) \\ c_2, \hat{c_2} \epsilon C(n_2) \\ c_1.w = c_2.w \\ \hat{c_1}.w = \hat{c_2}.w}} \left(\begin{array}{c} 1 + CDP(c_1, c2) + CDP(\hat{c_1}, \hat{c_2}) + \\ CDP(c_1, c_2) * CDP(\hat{c_1}, \hat{c_2}) \end{array} \right)$$

If the two nodes are not equal then the number of common paths that peak at them will be zero. If the nodes are equal, then all the common downward paths between them will also be the paths that peak at them, hence it is the first term in the above equation. Next, the remaining paths that peak at them can be counted by considering every pair of common children nodes represented by c_1 & c_2 and $\hat{c_1}$ & $\hat{c_2}$. For example, for the common node 'was' in Figures 4 and 5, the children nodes 'cat' and 'chased' are common. The path 'cat → was ← chased' is a path that peaks at 'was', hence 1 is added in the second

term. All the downward paths from 'cat' when incremented up to 'was' and down to 'chased' are also the paths that peak at 'was' (namely 'a → cat → was ← chased'). Similarly, all the downward paths from 'chased' when incremented up to 'was' and down to 'cat' are also paths that peak at 'was' ('cat → was ← chased ← by', 'cat → was ← chased ← by ← dog', etc.). Hence the next two terms are present in the equation. Finally, all the downward paths from 'cat' when incremented up to 'was' and down to every downward path from 'chased' are also the paths that peak at 'was' ('a → cat → was ← chased ← by', 'a → cat → was ← chased ← by ← dog' etc.). Hence there is the product term present in the equation. It is important not to re-count a path from the opposite direction hence the two pairs of common children are considered only once (i.e. not reconsidered symmetrically).

The dependency word subsequence kernel between two dependency trees T_1 and T_2 is then simply:

$$K(T_1, T_2) = \sum_{\substack{n_1 \epsilon T_1 \\ n_2 \epsilon T_2 \\ n_1.w = n_2.w}} (1 + CPP(n_1, n_2))$$

We also want to count the number of common words between the two trees in addition to the number of common paths, hence 1 is added in the equation. The kernel is normalized to remove any bias due to different tree sizes:

$$K_{normalized}(T_1, T_2) = \frac{K(T_1, T_2)}{\sqrt{K(T_1, T_1) * K(T_2, T_2)}}$$

Since for any long path common between two trees, there will be many shorter paths within it which will be also common between the two trees, it is reasonable to downweight the contribution of long paths. We do this by introducing a parameter $\alpha \epsilon (0, 1]$ and by downweighting a path of length l by α^l. A similar mechanism was also used in the syntactic tree kernel (Collins and Duffy, 2001).

The equations for computing CDP and CPP are accordingly modified as follows to accommodate this downweighting.

$$CDP(n_1, n_2) = 0 \; if \; n_1.w \neq n_2.w$$

otherwise,

$$CDP(n_1, n_2) = \sum_{\substack{c_1 \epsilon C(n_1) \\ c_2 \epsilon C(n_2) \\ c_1.w = c_2.w}} (\alpha + \alpha * CDP(c_1, c_2))$$

$$CPP(n_1, n_2) = 0 \; if \; n_1.w \neq n_2.w$$

otherwise,

$$CPP(n_1, n_2) = CDP(n_1, n_2) + \sum_{\substack{c_1, \hat{c_1} \epsilon C(n_1) \\ c_2, \hat{c_2} \epsilon C(n_2) \\ c_1.w = c_2.w \\ \hat{c_1}.w = \hat{c_2}.w}} \left(\begin{array}{c} \alpha^2 + \alpha * CDP(c_1, c2) + \\ \alpha * CDP(\hat{c_1}, \hat{c_2}) + \\ \alpha^2 * CDP(c_1, c_2) * CDP(\hat{c_1}, \hat{c_2}) \end{array} \right)$$

This algorithm to compute all the common paths between two trees has worst time complexity of $O(|T_1||T_2|)$, where $|T_1|$ and $|T_2|$ are the number of nodes of the two trees T_1 and T_2 respectively. This is because CDP computations are needed for every pairs of nodes between the two trees and is recursively computed. Using dynamic programming their recomputations can be easily avoided. The CPP computations then simply add the CDP values[2]. If the nodes common between the two trees are sparse then the algorithm will run much faster. Since the algorithm only needs to store the CDP values, its space complexity is $O(|T_1||T_2|)$. Also note that this algorithm computes the number of common paths of *all* lengths unlike the word subsequence kernel in which the maximum subsequence length needs to be specified and the time complexity then depends on this length.

4 Related Work

Several modifications to the syntactic tree kernels have been proposed to overcome the type of problems pointed out in Subsection 2.2. Zhang et al. (2007) proposed a grammar-driven syntactic tree kernel which allows soft matching between the subtrees of the trees if that is deemed appropriate by the grammar. For example, their kernel will be able

[2]This analysis uses the fact that any node in a tree on average has $O(1)$ number of children.

to match the subtrees (NP (DT a) (NN cat)) and (NP (DT a) (JJ fat) (NN cat)) with some penalty. Moschitti (2006b) proposed a partial tree kernel which can partially match subtrees. Moschitti et al. (2007) proposed a tree kernel over predicate-argument structures of sentences based on the PropBank labels. Che et al. (2006) presented a hybrid tree kernel which combines a constituent and a path kernel. We however note that the paths in this kernel link predicates and their arguments and are very different from general paths in a tree that our dependency-based word subsequence kernel uses. Shen et al. (2003) proposed a lexicalized syntactic tree kernel which utilizes LTAG-based features. Toutanova et al. (2004) compute similarity between two HPSG parse trees by finding similarity between the leaf projection paths using string kernels.

A few kernels based on dependency trees have also been proposed. Zelenko et al. (2003) proposed a tree kernel over shallow parse tree representations of sentences. This tree kernel was slightly generalized by Culotta and Sorensen (2004) to compute similarity between two dependency trees. In addition to the words, this kernel also incorporates word classes into the kernel. The kernel is based on counting matching subsequences of children of matching nodes. But as was also noted in (Bunescu and Mooney, 2005a), this kernel is *opaque* i.e. it is not obvious what the implicit features are and the authors do not describe it either. In contrast, our dependency-based word subsequence kernel, which also computes similarity between two dependency trees, is very transparent with the implicit features being simply the dependency paths. Their kernel is also very time consuming and in their more general sparse setting it requires $O(mn^3)$ time and $O(mn^2)$ space, where m and n are the number of nodes of the two trees ($m >= n$) (Zelenko et al., 2003).

Bunescu and Mooney (2005a) give a shortest path dependency kernel for relation extraction. Their kernel, however, does not find similarity between two sentences but between the shortest dependency paths connecting the two entities of interests in the sentences. This kernel uses general dependency graphs but if the graph is a tree then the shortest path is the only path between the entities. Their kernel also uses word classes in addition to the words themselves.

5 Experiments

We show that the new dependency-based word subsequence kernel performs better than word subsequence kernel and syntactic tree kernel on the task of domain-specific semantic parsing.

5.1 Semantic Parsing

Semantic parsing is the task of converting natural language sentences into their domain-specific complete formal meaning representations which an application can execute, for example, to answer database queries or to control a robot. A learning system for semantic parsing induces a semantic parser from the training data of natural language sentences paired with their respective meaning representations. KRISP (Kate and Mooney, 2006) is a semantic parser learning system which uses word subsequence kernel based SVM (Cristianini and Shawe-Taylor, 2000) classifiers and was shown to be robust to noise compared to other semantic parser learners. The system learns an SVM classifier for every production of the meaning representation grammar which tells the probability with which a substring of the sentence represents the semantic concept of the production. Using these classifiers a complete meaning representation of an input sentence is obtained by finding the most probable parse which covers the whole sentence. For details please refer to (Kate and Mooney, 2006).

The key operation in KRISP is to find the similarity between any two substrings of two natural language sentences. Word subsequence kernel was employed in (Kate and Mooney, 2006) to compute the similarity between two substrings. We modified KRISP so that the similarity between two substrings can also be computed using the syntactic tree kernel and the dependency-based word subsequence kernel. For applying the syntactic tree kernel, the syntactic subtree over a substring of a sentence is determined from the syntactic tree of the sentence by finding the lowest common ancestor of the words in this substring and then considering the smallest subtree rooted at this node which includes all the words of the substring. For applying the dependency-based word subsequence kernel to two substrings of a sentence, the kernel computation was suitably modified so that the common paths between the two depen-

dency trees always begin and end with the words present in the substrings. This is achieved by including only those downward paths in computations of CDP which end with words within the given substrings. These paths relate the words within the substrings perhaps using words outside of these substrings.

5.2 Methodology

We measure the performance of KRISP obtained using the three types of kernels on the GEOQUERY corpus which has been used previously by several semantic parsing learning systems. It contains 880 natural language questions about the US geography paired with their executable meaning representations in a functional query language (Kate et al., 2005). Since the purpose of the experiments is to compare different kernels and not different semantic parsers, we do not compare the performance with other semantic parser learning systems. The training and testing was done using standard 10-fold cross-validation and the performance was measured in terms of precision (the percentage of generated meaning representations that were correct) and recall (the percentage of all sentences for which correct meaning representations were obtained). Since KRISP assigns confidences to the meaning representations it outputs, an entire range of precision-recall trade-off can be obtained. We measure the best F-measure (harmonic mean of precision and recall) obtained when the system is trained using increasing amounts of training data.

Since we were not interested in the accuracy of dependency trees or syntactic trees but in the comparison between various kernels, we worked with gold-standard syntactic trees. We did not have gold-standard dependency trees available for this corpus so we obtained them indirectly from the gold-standard syntactic trees using the head-rules from (Collins, 1999). We however note that accurate syntactic trees can be obtained by training a syntactic parser on WSJ treebank and gold-standard parse trees of some domain-specific sentences (Kate et al., 2005).

In the experiments, the α parameter of the dependency-based word subsequence kernel was set to 0.25, the λ parameter of the word subsequence kernel was fixed to 0.75 and the downweighting parameter for the syntactic tree kernel was fixed to 0.4. These were determined through pilot experiments with a smaller portion of the data set. The maximum length of subsequences required by the word subsequence kernel was fixed to 3, a longer length was not found to improve the performance and was only increasing the running time.

Examples	Dependency	Word	Syntactic
40	**25.62**	21.51	23.65
80	**45.30**	42.77	43.14
160	**63.78**	61.22	59.66
320	**72.44**	70.36	67.05
640	77.32	77.82	74.26
792	79.79	79.09	76.62

Table 1: Results on the semantic parsing task with increasing number of training examples using dependency-based word subsequence kernel, word subsequence kernel and syntactic tree kernel.

5.3 Results

Table 1 shows the results. The dependency-based word subsequence kernel always performs better than the syntactic tree kernel. All the numbers under the dependency kernel were found statistically significant ($p < 0.05$) over the corresponding numbers under the syntactic tree kernel based on paired t-tests. The improvement of the dependency-based word subsequence kernel over the word subsequence kernel is greater with less training data, showing that the dependency information is more useful when the training data is limited. The performance converges with higher amounts of training data. The numbers shown in bold were found statistically significant over the corresponding numbers under the word subsequence kernel.

It may be noted that syntactic tree kernel is mostly doing worse than the word subsequence kernel. We believe this is because of the shortcomings of the syntactic tree kernel pointed out in Subsection 2.2. Since this is a semantic processing task, the words play an important role and the generalized syntactic categories are not very helpful.

6 Future Work

In future, the dependency-based word subsequence kernel could be extended to incorporate word classes

like the kernels presented in (Bunescu and Mooney, 2005a; Zelenko et al., 2003). It should be possible to achieve this by incorporating matches between word classes in addition to the exact word matches in the kernel computations similar to the way in which the word subsequence kernel was extended to incorporate word classes in (Bunescu and Mooney, 2005b). This will generalize the kernel and make it more robust to data sparsity.

The dependency-based word subsequence kernel could be tested on other tasks which require computing similarity between sentences or texts, like text classification, paraphrasing, summarization etc. We believe this kernel will help improve performance on those tasks.

7 Conclusions

We introduced a new kernel which finds similarity between two sentences as the number of common paths shared between their dependency trees. This kernel can also be looked upon as an improved word subsequence kernels which only counts the common word subsequences which are related by dependencies. We also gave an efficient algorithm to compute this kernel. The kernel was shown to out-perform the word subsequence kernel and the syntactic tree kernel on the task of semantic parsing.

References

Razvan C. Bunescu and Raymond J. Mooney. 2005a. A shortest path dependency kernel for relation extraction. In *Proc. of HLT/EMNLP-05*, pages 724–731, Vancouver, BC, October.

Razvan C. Bunescu and Raymond J. Mooney. 2005b. Subsequence kernels for relation extraction. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, Vancouver, BC.

Nicola Cancedda, Eric Gaussier, Cyril Goutte, and Jean Michel Renders. 2003. Word sequence kernels. *Journal of Machine Learning Research, Special Issue on Machine Learning Methods for Text and Images*, 3:1059–1082, February.

Wanxiang Che, Min Zhang, Ting Liu, and Sheng Li. 2006. A hybrid convolution tree kernel for semantic role labeling. In *Proc. of COLING/ACL-06*, pages 73–80, Sydney, Australia, July.

Michael Collins and Nigel Duffy. 2001. Convolution kernels for natural language. In *Proc. of NIPS-2001*.

Michael Collins. 1999. *Head-driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania.

Michael Collins. 2002. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proc. of ACL-2002*, pages 263–270, Philadelphia, PA, July.

Nello Cristianini and John Shawe-Taylor. 2000. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press.

Aron Culotta and Jeffrey Sorensen. 2004. Dependency tree kernels for relation extraction. In *Proc. of ACL-04*, pages 423–429, Barcelona, Spain, July.

Richard Hudson. 1984. *Word Grammar*. Blackwell.

Rohit J. Kate and Raymond J. Mooney. 2006. Using string-kernels for learning semantic parsers. In *Proc. of COLING/ACl-06*, pages 913–920, Sydney, Australia, July.

Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. 2005. Learning to transform natural to formal languages. In *Proc. AAAI-2005*, pages 1062–1068, Pittsburgh, PA, July.

Terry Koo, Xavier Carreras, and Michael Collins. 2008. Simple semi-supervised dependency parsing. In *Proc. of ACL-08*, pages 595–603, Columbus, Ohio, June.

Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. 2002. Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proc. of HLT/EMNLP-05*, pages 523–530, Vancouver, BC.

Alessandro Moschitti, Silvia Quarteroni, Roberto Basili, and Suresh Manandhar. 2007. Exploiting syntactic and shallow semantic kernels for question answer classification. In *Proc. of ACL-07*, pages 776–783, Prague, Czech Republic, June.

Alessandro Moschitti. 2006a. Making tree kernels practical for natural language learning. In *Proc. of EACL-06*, pages 113–120, Trento, Italy, April.

Alessandro Moschitti. 2006b. Syntactic kernels for natural language learning: the semantic role labeling case. In *Proc. of HLT/NAACL-06, short papers*, pages 97–100, New York City, USA, June.

Juho Rousu and John Shawe-Taylor. 2005. Efficient computation of gapped substring kernels on large alphabets. *Journal of Machine Learning Research*, 6:1323–1344.

Libin Shen, Anoop Sarkar, and Aravind Joshi. 2003. Using ltag based features in parse reranking. In *Proc. of EMNLP-2003*, pages 89–96, Sapporo, Japan, July.

Kristina Toutanova, Penka Markova, and Christopher Manning. 2004. The leaf projection path view of parse trees: Exploring string kernels for HPSG parse selection. In *Proc. EMNLP-04*, pages 166–173, Barcelona, Spain, July.

Vladimir N. Vapnik. 1998. *Statistical Learning Theory*. John Wiley & Sons.

Qin Iris Wang, Dale Schuurmans, and Dekang Lin. 2008. Semi-supervised convex training for dependency parsing. In *Proceedings of ACL-08: HLT*, pages 532–540, Columbus, Ohio, June.

D. Zelenko, C. Aone, and A. Richardella. 2003. Kernel methods for relation extraction. *Journal of Machine Learning Research*, 3:1083–1106.

Min Zhang, Wanxiang Che, Aiti Aw, Chew Lim Tan, Guodong Zhou, Ting Liu, and Sheng Li. 2007. A grammar-driven convolution tree kernel for semantic role classification. In *Proc. of ACL-2007*, pages 200–207, Prague, Czech Republic, June.