Speeding up Training with Tree Kernels for Node Relation Labeling

Jun'ichi Kazama and **Kentaro Torisawa**
Japan Advanced Institute of Science and Technology (JAIST)
Asahidai 1-1, Nomi, Ishikawa, 923-1292 Japan
{kazama, torisawa}@jaist.ac.jp

Abstract

We present a method for speeding up the calculation of tree kernels during training. The calculation of tree kernels is still heavy even with efficient dynamic programming (DP) procedures. Our method maps trees into a small feature space where the inner product, which can be calculated much faster, yields the same value as the tree kernel for *most* tree pairs. The training is sped up by using the DP procedure only for the exceptional pairs. We describe an algorithm that detects such exceptional pairs and converts trees into vectors in a feature space. We propose tree kernels on *marked labeled ordered trees* and show that the training of SVMs for semantic role labeling using these kernels can be sped up by a factor of several tens.

1 Introduction

Many NLP tasks such as parse selection and tagging can be posed as the classification of labeled ordered trees. Several tree kernels have been proposed for building accurate kernel-based classifiers (Collins and Duffy, 2001; Kashima and Koyanagi, 2002). They have the following form in common.

$$K(T_1, T_2) = \sum_{S_i} W(S_i) \cdot \#_{S_i}(T_1) \cdot \#_{S_i}(T_2), \quad (1)$$

where S_i is a possible subtree, $\#_{S_i}(T_j)$ is the number of times S_i is *included* in T_j, and $W(S_i)$ is the weight of S_i. That is, tree kernels are inner products in a subtree feature space where a tree is mapped to vector $V(T_j) = \left(\sqrt{W(S_i)}\#_{S_i}(T_j)\right)_i$. With tree kernels we can take global structures into account, while alleviating overfitting with kernel-based learning methods such as support vector machines (SVMs) (Vapnik, 1995).

Previous studies (Collins and Duffy, 2001; Kashima and Koyanagi, 2002) showed that although it is difficult to explicitly calculate the inner product in Eq. (1) because we need to consider an exponential number of possible subtrees, the tree kernels can be computed in $O(|T_1||T_2|)$ time by using dynamic programming (DP) procedures. However, these DP procedures are time-consuming in practice.

In this paper, we present a method for speeding up the training with tree kernels. Our target application is *node relation labeling*, which includes NLP tasks such as semantic role labeling (SRL) (Gildea and Jurafsky, 2002; Moschitti, 2004; Hacioglu et al., 2004). For this purpose, we designed kernels on *marked labeled ordered trees* and derived $O(|T_1||T_2|)$ procedures. However, the lengthy training due to the cost of kernel calculation prevented us from assessing the performance of these kernels and motivated us to make the training practically fast.

Our speed-up method is based on the observation that very few pairs in the training set have a great many common subtrees (we call such pairs *malicious* pairs) and most pairs have a very small number of common subtrees. This leads to a drastic variance in kernel values, e.g., when $W(S_i) = 1$. We thus call this property of data *unbalanced similarity*. Fast calculation based on the inner product is possible for non-malicious pairs since we can convert the trees into vectors in a space of a small subset of all subtrees. We can speed up the training by using the DP procedure only for the rare malicious pairs.

We developed the FREQTM algorithm, a modification of the FREQT algorithm (Asai et al., 2002), to detect the malicious pairs and efficiently convert trees into vectors by enumerating only the subtrees actually needed (feature subtrees). The experiments demonstrated that our method makes the training of SVMs for the SRL task faster by a factor of several tens, and that it enables the performance of the kernels to be assessed in detail.

2 Kernels for Labeled Ordered Trees

The tree kernels proposed so far differ in how subtree inclusion is defined. For instance, Kashima and Koyanagi (2002) used the following definition.

DEFINITION 2.1 *S is included in T iff there exists a one-to-one function ψ from a node of S to a node of T such that (i) $pa(\psi(n_i)) = \psi(pa(n_i))$ ($pa(n_i)$ returns the parent of node n_i), (ii) $\psi(n_i) \succeq \psi(n_j)$ iff $n_i \succeq n_j$ ($n_i \succeq n_j$ means that n_i is an elder sibling of n_j), and (iii) $l(\psi(n_i)) = l(n_i)$ ($l(n_i)$ returns the label of n_i).*

We refer to the tree kernel based on this definition as K_{lo}. Collins and Duffy (2001) used a more restrictive definition where the preservation of CFG productions, i.e., $nc(\psi(n_i)) = nc(n_i)$ if $nc(n_i) > 0$ ($nc(n_i)$ is the number of children of n_i), is required in addition to the requirements in Definition 2.1. We refer to the tree kernel based on this definition as K_c.

It is pointed that extremely unbalanced kernel values cause overfitting. Therefore, Collins and Duffy (2001) used $W(S_i) = \lambda^{(\# \text{ of productions in } S_i)}$, and Kashima and Koyanagi (2002) used $W(S_i) = \lambda^{|S_i|}$, where λ ($0 \leq \lambda \leq 1$) is a factor to alleviate the unbalance by penalizing large subtrees.

To calculate the tree kernels efficiently, Collins and Duffy (2001) presented an $O(|T_1||T_2|)$ DP procedure for K_c. Kashima and Koyanagi (2002) presented one for K_{lo}. The point of these procedures is that Eq. (1) can be transformed:

$$K(T_1, T_2) = \sum_{n_1 \in T_1} \sum_{n_2 \in T_2} C(n_1, n_2),$$
$$C(n_1, n_2) \equiv \sum_{S_i} W(S_i) \cdot \#_{S_i}(T_1 \vartriangle n_1) \cdot \#_{S_i}(T_2 \vartriangle n_2),$$

where $\#_{S_i}(T_j \vartriangle n_k)$ is the number of times S_i is included in T_j with $\psi(root(S_i)) = n_k$. $C(n_1, n_2)$ can then be calculated recursively from those of the children of n_1 and n_2.

3 Kernels for Marked Labeled Ordered Trees for Node Relation Labeling

3.1 Node Relation Labeling

The node relation labeling finds relations among nodes in a tree. Figure 1 illustrates the concept of node relation labeling with the SRL task as an example. A0, A1, and AM-LOC are the semantic roles of the arguments of the verb "see (saw)". We represent an argument by the node that is the highest in the parse tree among the nodes that exactly cover the words in the argument. The node for the verb is determined similarly. For example, the node labeled "PP" represents the AM-LOC argument "in the sky", and the node labeled "VBD" represents the verb "see (saw)". We assume that there is a two-node relation labeled with the semantic role (represented by the arrow in the figure) between the verb node and the argument node.

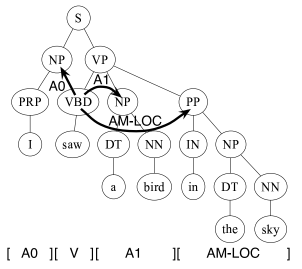

Figure 1: Node relation labeling.

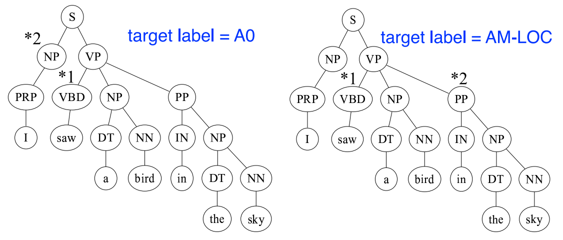

Figure 2: Semantic roles encoded by marked labeled ordered trees.

3.2 Kernels on Marked Labeled Ordered Trees

We define a marked labeled ordered tree as a labeled ordered tree in which each node has a mark in addition to a label. We use $m(n_i)$ to denote the mark of node n_i. If n_i has no mark, $m(n_i)$ returns the special mark no-mark. We also use the function $marked(n_i)$, which returns $true$ iff $m(n_i)$ is not no-mark. We can encode a k-node relation by using k distinct marks. Figure 2 shows how the semantic roles illustrated in Figure 1 can be encoded using marked labeled ordered trees. We used the mark *1 to represent the verb node and *2 to represent the argument node.

The node relation labeling task can be posed as the classification of marked trees that returns $+1$ when the marks encode the correct relation and -1

Algorithm 3.1: KERNELLOMARK(T_1, T_2)

```
(nodes are ordered by the post-order traversal)
for $n_1 \leftarrow 1$ to $|T_1|$ do
  for $n_2 \leftarrow 1$ to $|T_2|$ do ———————— (A)
    if $lm(n_1) \neq lm(n_2)$ then
      $C(n_1, n_2) \leftarrow 0$  $C^r(n_1, n_2) \leftarrow 0$
    else if $n_1$ and $n_2$ are leaf nodes then
      $C(n_1, n_2) \leftarrow \lambda$
      if $marked(n_1)$ and $marked(n_2)$ then
        $C^r(n_1, n_2) \leftarrow \lambda$  else $C^r(n_1, n_2) \leftarrow 0$
    else
      $S(0, j) \leftarrow 1$  $S(i, 0) \leftarrow 1$
      if $marked(n_1)$ and $marked(n_2)$ then
        $S^r(0, j) \leftarrow 1$  $S^r(i, 0) \leftarrow 1$
      else $S^r(0, j) \leftarrow 0$  $S^r(i, 0) \leftarrow 0$
      for $i \leftarrow 1$ to $nc(n_1)$ do
        for $j \leftarrow 1$ to $nc(n_2)$ do
          $S(i, j) \leftarrow$
            $S(i-1, j) + S(i, j-1) - S(i-1, j-1)$
            $+S(i-1, j-1) \cdot C(ch_i(n_1), ch_j(n_2))$
          $S^r(i, j) \leftarrow$ ———————— (B)
            $S^r(i-1, j) + S^r(i, j-1) - S^r(i-1, j-1)$
            $+S^r(i-1, j-1) \cdot C(ch_i(n_1), ch_j(n_2))$
            $+S(i-1, j-1) \cdot C^r(ch_i(n_1), ch_j(n_2))$
            $-S^r(i-1, j-1) \cdot C^r(ch_i(n_1), ch_j(n_2))$
      $C(n_1, n_2) \leftarrow \lambda \cdot S(nc(n_1), nc(n_2))$
      $C^r(n_1, n_2) \leftarrow \lambda \cdot S^r(nc(n_1), nc(n_2))$
return $(\sum_{n_1=1}^{|T_1|} \sum_{n_2=1}^{|T_2|} C^r(n_1, n_2))$
```

otherwise. To enable such classification, we need tree kernels that take into account the node marks. We thus propose mark-aware tree kernels formulated as follows.

$$K(T_1, T_2) = \sum_{S_i : marked(S_i)} W(S_i) \cdot \#_{S_i}(T_1) \cdot \#_{S_i}(T_2),$$

where $marked(S_i)$ returns $true$ iff $marked(n_i) = true$ for at least one node in tree S_i. In these kernels, we require $m(\psi(n_i)) = m(n_i)$ in addition to $l(\psi(n_i)) = l(n_i)$ for subtree S_i to be regarded as included in tree T_j. In other words, these kernels treat $lm(n_i) \equiv (l(n_i), m(n_i))$ as the new label of node n_i and sum only over subtrees that have at least one marked node. We refer to the marked version of K_{lo} as K_{lo}^r and the marked version of K_c as K_c^r.

We can derive $O(|T_1||T_2|)$ DP procedures for the above kernels as well. Algorithm 3.1 shows the DP procedure for K_{lo}^r, which is derived by extending the DP procedure for K_{lo} (Kashima and Koyanagi, 2002). The key is the use of $C^r(n_1, n_2)$, which stores the sum over only marked subtrees, and its recursive calculation using $C(n_1, n_2)$ and $C^r(n_1, n_2)$ **(B)**. An $O(|T_1||T_2|)$ procedure for K_c^r can also be derived by extending (Collins and Duffy, 2001).

Table 1: Malicious and non-malicious pairs in the 1k data (3,136 trees) used in Sec. 5.2. We used $K(T_i, T_j) = 10^4$ with $\lambda = 1$ as the threshold for maliciousness. (A): pairs (i, i). (B): pairs from the same sentence except (i, i). (C): pairs from different sentences. Some malicious pairs are from different but similar sentences, which are difficult to detect.

		K_{lo}^r		K_c^r	
		# pairs	avg. $K(T_i, T_j)$	# of pairs	avg. $K(T_i, T_j)$
$\geq 10^4$	(A)	3,121	1.17×10^{52}	3,052	2.49×10^{32}
	(B)	7,548	7.24×10^{48}	876	1.26×10^{32}
	(C)	6,510	6.80×10^{9}	28	1.82×10^{4}
$< 10^4$	(A)	15	4.19×10^{3}	84	3.06×10^{3}
	(B)	4,864	2.90×10^{2}	11,536	1.27×10^{2}
	(C)	9,812,438	1.82×10^{1}	9,818,920	1.84×10^{-1}

4 Fast Training with Tree Kernels

4.1 Basic Idea

As mentioned, we define two types of tree pairs: malicious and non-malicious pairs. Table 1 shows how these two types of pairs are distributed in an actual training set. There is a clear distinction between malicious and non-malicious pairs, and we can exploit this property to speed up the training.

We define subset $\mathcal{F} = \{F_i\}$ (*feature subtrees*), which includes only the subtrees that appear as a common included subtree in the non-malicious pairs. We convert a tree to feature vector $V(T_j) = \left(\sqrt{W(F_i)}\#_{F_i}(T_j)\right)_i$ using only $\mathcal{F}$. Then we use a procedure that chooses the DP procedure or the inner product procedure depending on maliciousness:

$$K(T_i, T_j) = \begin{cases} K(T_i, T_j) \text{ (DP)} & \text{if } (i, j) \text{ is malicious.} \\ \langle V(T_i), V(T_j) \rangle & \text{otherwise} \end{cases}$$

This procedure returns the same value as the original calculation. Naively, if $|V(T_i)|$ (the number of feature subtrees such that $\#_{F_i}(T_i) \neq 0$) is small enough, we can expect a speed-up because the cost of calculating the inner product is $O(|V(T_i)| + |V(T_j)|)$. However, since $|V(T_i)|$ might increase as the training set becomes larger, we need a way to scale the speed-up to large data. In most kernel-based methods, such as SVMs, we actually need to calculate the kernel values with all the training examples for a given example T_i: $KS(T_i) = \{K(T_i, T_1), \ldots, K(T_i, T_L)\}$, where L is the number of training examples. Using *occurrence pattern* $OP(F_i) = \{(k, \#_{F_i}(T_k)) | \#_{F_i}(T_k) \neq 0\}$ pre-

Algorithm 4.1: CALCULATEKS(T_i)

```
for each $F$ such that $\#_F(T_i) \neq 0$ do
  for each $(j, \#_F(T_j)) \in OP(F)$ do
    $KS(j) \leftarrow KS(j) + W(F) \cdot \#_F(T_i) \cdot \#_F(T_j)$    (A)
for $j = 1$ to $L$ do
    if $(i, j)$ is malicious then $KS(j) \leftarrow K(T_i, T_j)$  (DP)
```

pared beforehand, we can calculate $KS(T_i)$ efficiently (Algorithm 4.1). A similar technique was used in (Kudo and Matsumoto, 2003a) to speed up the calculation of inner products.

We can show that the per-pair cost of Algorithm 4.1 is $O(c_1 Q + r_m c_2 |T_i||T_j|)$, where Q is the average number of common feature subtrees in a tree pair, r_m is the rate of malicious pairs, c_1 and c_2 are the constant factors for vector operations and DP operations. This cost is independent of the number of training examples. We expect from our observations that both Q and r_m are very small and that $c_1 \ll c_2$.

4.2 Feature Subtree Enumeration with Malicious Pair Detection

To detect malicious pairs and enumerate feature subtrees $\mathcal{F}$ (and to convert each tree to a feature vector), we developed an algorithm based on the FREQT algorithm (Asai et al., 2002). The FREQT algorithm can efficiently enumerate subtrees that are included (Definition 2.1) in more than a pre-specified number of trees in the training examples by generating candidate subtrees using *right most expansions* (*RMEs*). FREQT-based algorithms have recently been used in methods that treat subtrees as features (Kudo and Matsumoto, 2004; Kudo and Matsumoto, 2003b).

To develop the algorithm, we made the definition of maliciousness more search-oriented since it is costly to check for maliciousness based on the exact number of common subtrees or the kernel values (i.e., by using the DP procedure for all L^2 pairs). Whatever definition we use, the correctness is preserved as long as we do not fail to enumerate the subtrees that appear in the pairs we consider non-malicious. First, we consider pairs (i, i) to always be malicious. Then, we use a FREQT search that enumerates the subtrees that are included in at least two trees as a basis. Next, we modify FREQT so that it stops the search if candidate subtree F_i is too large (larger than size D, e.g., 20), and we regard the pairs of the trees where F_i appears as malicious because having a large subtree in common implies having a

Algorithm 4.2: FREQTM(D, R)

```
procedure GENERATECANDIDATE($F_i$)
  for each $(j, n) \in occ(F_i)$ do
    for each $(F_k, n_r) \in RME(F_i, T_j, n)$ do
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{F_k\}$;  $occ(F_k) \leftarrow occ(F_k) \cup (j, n_r)$
      $\underline{\textbf{if } |occ(F_k)|/|sup(F_i)| > R \textbf{ then}}$
        $\underline{\textbf{return } ((\phi, \textbf{ false }))}$————————(R)
  return $((\{F_k | F_k \in \mathcal{S}, |sup(F_k)| \geq \underline{2}\}, \underline{\textbf{ true }}))$

procedure SEARCH($F_i, \underline{precheck}$)
  $\underline{\textbf{if } |F_i| \geq D \textbf{ then } \text{REGISTERMAL}(F_i) \textbf{ return ( false )}}$–(P)
  $\underline{(\mathcal{C}, suc) \leftarrow \text{GENERATECANDIDATE}(F_i)}$
  $\underline{\textbf{if not } suc \textbf{ then } \text{REGISTERMAL}(F_i) \textbf{ return ( false )}}$—(S)
  for each $F_k \in \mathcal{C}$ do
    $\underline{\textbf{if } malicious(F_k) \textbf{ then goto next } F_k}$   ————-(P2)
    $\underline{suc \leftarrow} \text{SEARCH}(F_k, \underline{precheck})$
    $\underline{\textbf{if not } suc \textbf{ and } |sup(F_i)| = |sup(F_k)| \textbf{ then}}$
      $\underline{\textbf{return ( false )}}$——————————(P1)
  if $\underline{\textbf{not } precheck \textbf{ and } marked(F_i)}$ then
    REGISTERSUBTREE($F_i$)————————————(F)
  $\underline{\textbf{return ( true )}}$

main
  $\underline{\mathcal{M} \leftarrow \phi}$  (a set of malicious pairs)
  $\mathcal{F}^1 \leftarrow \{F_i | |F_i| = 1 \text{ and } |sup(F_i)| \geq \underline{2}\}$
  $\underline{\textbf{for each } F_i \in \mathcal{F}^1 \textbf{ do } \text{SEARCH}(F_i, \textbf{ true })}$————(PC)
  for each $F_i \in \mathcal{F}^1$ do SEARCH($F_i$, $\underline{\textbf{false}}$ )
  $\underline{\mathcal{M} \leftarrow \mathcal{M} \cup \{(i, i) | 1 \leq i \leq l\}}$
  return $(\underline{\mathcal{M}}, \{V(T_i)\}, \underline{\{W(f_i)\}})$
```

Table 2: Functions in FREQTM.

- $occ(F_i)$ returns occurrence list of F_i whose element (j, n) indicates that F_i appears in T_j and that n (of T_j) is the node added to generated F_i in T_j by the RME (n works as the position of F_i in T_j).
- $sup(F_i)$ returns the IDs of distinct trees in $occ(F_i)$.
- $malicious(F_i)$ returns $true$ iff all pairs in $sup(F_i)$ are already registered in the set of malicious pairs, $\mathcal{M}$. (Currently, this returns $false$ if $|sup(F_i)| > M$ where M is the maximum support size of the malicious subtrees so far. We will remove this check since we found that it did not affect efficiency so much.)
- $RME(F_i, T_j, n)$ is a set of subtrees generated by RMEs of F_i in T_j (permitted when previously expanded node to generate F_i is n).

possibly exponential number of subtrees of that subtree in common. Although this test is heuristic and conservative in that it ignores the shape and marks of a tree, it works fine empirically.

Algorithm 4.2 is our algorithm, which we call FREQTM. The differences from FREQT are underlined. Table 2 summarizes the functions used. To make the search efficient, pruning is performed as follows (see also Figure 3). The basic idea behind is that if $malicious(F_i)$ is $true$ then $malicious(F_k)$ is also $true$ for F_k that is expanded from F_i by an

RME since $sup(F_k) \subseteq sup(F_i)$. This means we do not need to enumerate F_i nor any descendant of F_i.

- **(P)** Once $|F_i| \geq D$ and the malicious pairs are registered, we stop searching further.
- **(P1)** If the search from F_k (expanded from F_i) found a malicious subtree and if $|sup(F_i)| = |sup(F_k)|$, we stop the search from any other subtree F_m (expanded from F_i) since we can prove that $malicious(F_m) = true$ without actually testing it (proof omitted).
- **(P2)** If $malicious(F_k) = true$, we prune the search from F_k. To prune even when $malicious(F_k)$ becomes $true$ as a result of succeeding searches, we first run a search only for detecting malicious pairs (see **(PC)**).
- **(S)** We stop searching when the occurrence list becomes too long (larger than threshold R) since it causes a severe search slowdown.

Note that we use a depth-first version of FREQT as a basis to first find the largest subtrees and to detect malicious pairs at early points in the search. Enumeration of unnecessary subtrees is avoided because the registration of subtrees is performed at the post-order position **(F)**. The conversion to vectors is performed by assigning an ID to subtree F_i when registering it at **(F)** and distributing the ID to all the examples in $occ(F_i)$. Finally, D should be large enough to make r_m sufficiently small but should not be so large that too many feature subtrees are enumerated.

We expect that the cost of FREQTM is offset by the faster training, especially when training on the same data is repeatedly performed as in the tuning of hyperparameters.

For K_c^r, we use a similar search procedure. However, the RME is modified so that all the children of a CFG production are expanded at once. Although the modification is not trivial, we omit the explanation due to space limitations.

4.3 Feature Compression

Additionally, we use a simple but effective feature compression technique to boost speed-up. The idea is simple: feature subtrees F_i and F_j can be treated as one feature f_k, with weight $W(f_k) = W(F_i) + W(F_j)$ if $OP(F_i) = OP(F_j)$. This drastically reduces the number of features. Although this is similar to finding closed and maximal subtrees (Chi et al., 2004), it is easy to implement since we need only the occurrence pattern, $OP(F_i)$, which is easily obtained from $occ(F_i)$ in the FREQTM search.

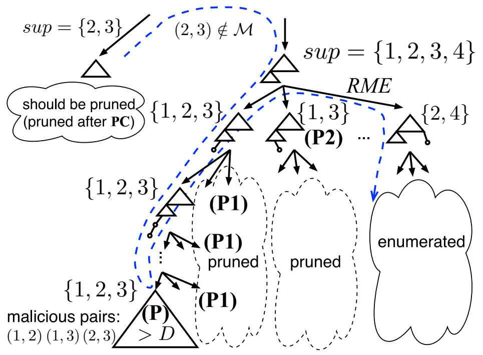

Figure 3: Pruning in FREQTM.

4.4 Alternative Methods

Vishwanathan and Smola (2004) presented the $O(|T_1| + |T_2|)$ procedure that exploits suffix trees to speed up the calculation of tree kernels. However, it can be applied to only a few types of subtrees that can be represented as a contiguous part in a string representation of a tree. Therefore, neither K_{lo}^r nor K_c^r can be sped up by using this procedure.

Another method is to change an inner loop, such as **(B)** in Algorithm 3.1, so that it iterates only over nodes in T_2 that have $l(n_1)$. We use this as the baseline for comparison, since we found that this is about two times faster than the standard implementation. [1]

4.5 Remaining Problem

Note that the method described here cannot speed up classification, since the converted vectors are valid only for calculating the kernels between trees in the training set. However, when we classify the same trees repeatedly, we can convert the trees in the training set and the classified trees at the same time and use the obtained vectors for classification.

5 Evaluation

We first evaluated the speed-up by our method for the semantic role labeling (SRL) task. We then demonstrated that the speed-up method enables a detailed comparison of K_{lo}^r and K_c^r for the SRL task.

[1]For K_c^r, it might be possible to speed up comparisons in the algorithm by assigning IDs for CFG rules. We leave this for future work since it complicates implementation.

Table 3: Conversion statistics and speed-up for semantic role A2.

	K^r_{lo}					K^r_c				
size (# positive examples)	1,000	2,000	4,000	8,000	12,000	1,000	2,000	4,000	8,000	12,000
# examples	3,136	6,246	12,521	25,034	34,632	3,136	6,246	12,521	25,034	34,632
# feature subtrees ($\times 10^4$)	804.4	2,427.3	6,542.9	16,750.1	26,146. 5	3.473	9.009	21.867	52.179	78.440
# features (compressed) ($\times 10^4$)	20.7	67.3	207.2	585.9	977.0	0.580	1.437	3.426	8.128	12.001
avg. $\|V\|$ (compressed)	468.0	866.5	1,517.3	2,460.5	3,278.3	10.5	14.0	17.9	23.1	25.9
rate of malicious pairs r_m (%)	0.845	0.711	0.598	0.575	1.24	0.161	0.0891	0.0541	0.0370	0.0361
conversion time (sec.)	208.0	629.2	1,921.1	6,519.8	14,824.9	3.8	8.7	20.4	46.5	70.4
SVM time (DP+lookup) (sec.)	487.9	1,716.2	4,526.4	79,800.7	92,542.2	360.7	1,263.5	5,893.3	53,055.5	47,089.2
SVM time (proposed) (sec.)	17.5	68.6	186.4	1,721.7	2,531.8	4.9	25.7	119.5	982.8	699.1
speed-up factor	**27.8**	**25.0**	**24.3**	**46.4**	**36.6**	**73.3**	**49.1**	**49.3**	**53.98**	**67.35**

5.1 Setting

We used the data set provided for the CoNLL05 SRL shared task (Carreras and Màrquez, 2005). We used only the training part and divided it into our training, development, and testing sets (23,899, 7,966, and 7,967 sentences, respectively). As the tree structure, we used the output of Collins' parser (with WSJ-style non-terminals) provided with the data set. We also used POS tags by inserting the nodes labeled by POS tags above the word nodes. The average number of nodes in a tree was about 82. We ignored any arguments (and verbs) that did not match any node in the tree (the rate of such cases was about 3.5%). [2] The words were lowercased.

We used TinySVM[3] as the implementation of SVM and added our tree kernels, K^r_{lo} and K^r_c. We implemented FREQTM based on the implementation of FREQT by Kudo.[4] We normalized the kernel values: $K(T_i, T_j)/\sqrt{K(T_i, T_i) \times K(T_j, T_j)}$. Note that this normalization barely affected the training time since we can calculate $K(T_i, T_i)$ beforehand.

We assumed two-step labeling where we first find the argument node and then we determine the label by using a binary classifier for each semantic role. In this experiment, we focused on the performance for the classifiers in the latter step. We used the marked labeled ordered tree that encoded the target role as a positive example and the trees that encoded other roles of the verb in the same sentence as negative examples. We trained and evaluated the classifiers using the examples generated as above. [5]

5.2 Training Speed-up

We calculated the statistics for the conversion by FREQTM and measured the speed-up in SVM training for semantic role A2, for various numbers of training examples. For FREQTM, we used $D = 20$ and $R = 20$. For SVM training, we used convergence tolerance 0.001 (-e option in TinySVM), soft margin cost $C = 1.0 \times 10^3$ (-c), maximum number of iterations 10^5, kernel cache size 512 MB (-m), and decay factor $\lambda = 0.2$ for the weight of each subtree. We compared the time with our fast method (Algorithm 4.1) with that with the DP procedure with the node lookup described in Section 4.4. Note that these two methods yield almost identical SVM models (there are very slight differences due to the numerical computation). The time was measured using a computer with 2.4-GHz Opterons.

Table 3 shows the results for K^r_{lo} and K^r_c. The proposed method made the SVM training substantially faster for both K^r_{lo} and K^r_c. As we expected, the speed-up factor did not decrease even though $|V|$ increased as the amount of data increased. Note that FREQTM actually detected non-trivial malicious pairs such as those from very similar sentences in addition to trivial ones, e.g., (i, i). FREQTM conversion was much faster and the converted feature vectors were much shorter for K^r_c, presumably because K^r_c restricts the subtrees more.

The compression technique described in Section 4.3 greatly reduced the number of features. Without this compression, the storage requirement would be impractical. It also boosted the speed-up. For example, the training time with K^r_{lo} for the size 1,000 data in Table 3 was 86.32 seconds without compression. This means that the compression boosted the

[2]This was caused by parse errors, which can be solved by using more accurate parsers, and by bracketing inconsistencies between parser outputs and SRL annotations (e.g., phrasal verbs), many of which can be avoided by using heuristic transformers.

[3]http://chasen.org/˜taku/software/TinySVM

[4]http://chasen.org/˜taku/software/freqt

[5]The evaluation is slightly easier since the classifier for role X is evaluated only on the examples generated from the sentences that contain a verb that has X as a role.

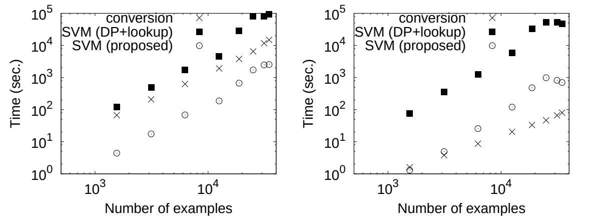

Figure 4: Scaling of conversion time and SVM training time. Left: K^r_{lo}. Right: K^r_c

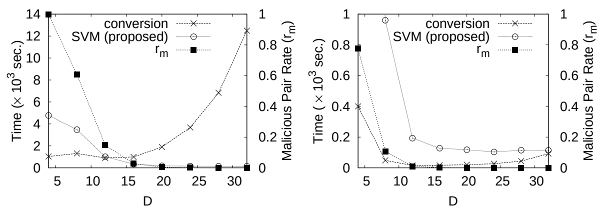

Figure 5: Relation between D and conversion time, SVM training time, and r_m. Left: K^r_{lo}. Right: K^r_c

speed-up by a factor of more than 5.

The cost of FREQTM is much smaller than that of SVM training with DP. Therefore, our method is beneficial even if we train the SVM only once.

To see how our method scales to large amounts of data, we plotted the time for the conversion and the SVM training w.r.t. data size on a log-log scale. As shown in Figure 4, the scaling factor was about 1.7 for the conversion time, 2.1 for SVM training with DP, and 2.0 for the proposed SVM training for K^r_{lo}. For K^r_c, the factors were about 1.3, 2.1, and 2.0, respectively. Regardless of the method, the cost of SVM training was about $O(L^2)$, as reported in the literature. Although FREQTM also has a superlinear cost, it is smaller than that of SVM training. Therefore, the cost of SVM training will become a problem before the cost of FREQTM does.

As we mentioned, the choice of D is a trade-off. Figure 5 shows the relationships between D and the time of conversion by FREQTM, the time of SVM training using the converted vectors, and the rate of malicious pairs, r_m. We can see that the choice of D is more important in the case of K_{lo} and that $D = 20$ used in our evaluation is not a bad choice.

5.3 Semantic Role Labeling

We assessed the performance of K^r_{lo} and K^r_c for semantic roles A1, A2, AM-ADV, and AM-LOC using our fast training method. We tuned soft margin cost C and λ by using the development set (we used the technique described in Section 4.5 to enable fast classification of the development set). We experimented with two training set sizes (4,000 and 8,000). For each λ (0.1, 0.15, 0.2, 0.25, and 0.30), we tested 40 different values of C ($C \in [2 \ldots 10^3]$ for size 4,000 and $C \in [0.5 \ldots 10^3]$ for size 8,000), and we evaluated the accuracy of the best setting for the test set.[6] Fast training is crucial since the performance differs substantially depending on the values of these hyperparameters. Table 4 shows the results. The accuracies are shown by F_1. We can see that K^r_{lo} outperformed K^r_c in all cases, presumably because K^r_c allows only too restrictive subtrees and therefore causes data sparseness. In addition, as one would expect, larger training sets are beneficial.

6 Discussion

The proposed speed-up method can also be applied to labeled ordered trees (e.g., for parse selection). However, the speed-up might be smaller since without node marks the number of subtrees increases while the DP procedure becomes simpler. On the other hand, the FREQTM conversion for marked labeled ordered trees might be made faster by exploiting the mark information for pruning. Although our method is not a complete solution in a classification setting, it might be in a clustering setting (in a sense it is training only). However, it is an open question whether unbalanced similarity, which is the key to our speed-up, is ubiquitous in NLP tasks and under what conditions our method scales better than the SVMs or other kernel-based methods.

Several studies claim that learning using tree kernels and other convolution kernels tends to overfit and propose selecting or restricting features (Cumby and Roth, 2003; Suzuki et al., 2004; Kudo and Matsumoto, 2004). Sometimes, the classification becomes faster as a result (Suzuki et al., 2004; Kudo and Matsumoto, 2004). We do not disagree with these studies. The fact that small λ values resulted in the highest accuracy in our experiment implies that too large subtrees are not so useful. However, since this tendency depends on the task, we need to assess the performance of full tree kernels for comparison. In this sense, our method is of great importance.

Node relation labeling is a generalization of node

[6]We used 10^6 as the maximum number of iterations.

Table 4: Comparison between K^r_{lo} and K^r_c.

		training set size = 4,000			training set size = 8,000		
		best setting	F_1 (dev)	F_1 (test)	best setting	F_1 (dev)	F_1 (test)
A1	K^r_{lo}	$\lambda = 0.2, C = 13.95$	**87.89**	**87.90**	$\lambda = 0.25, C = 8.647$	**89.80**	**89.81**
	K^r_c	$\lambda = 0.15, C = 3.947$	85.36	85.56	$\lambda = 0.2, C = 17.63$	87.93	87.96
A2	K^r_{lo}	$\lambda = 0.20, C = 13.95$	**85.65**	**84.70**	$\lambda = 0.20, C = 57.82$	**87.94**	**87.26**
	K^r_c	$\lambda = 0.10, C = 7.788$	84.79	83.51	$\lambda = 0.15, C = 1.0 \times 10^3$	87.37	86.23
AM-ADV	K^r_{lo}	$\lambda = 0.25, C = 8.647$	**86.20**	**86.64**	$\lambda = 0.15, C = 45.60$	**86.91**	**87.01**
	K^r_c	$\lambda = 0.20, C = 3.344$	83.58	83.72	$\lambda = 0.20, C = 2.371$	84.34	84.08
AM-LOC	K^r_{lo}	$\lambda = 0.15, C = 20.57$	**91.11**	**92.92**	N/A		
	K^r_c	$\lambda = 0.15, C = 13.95$	89.59	91.32	AM-LOC does not have more than 4,000 positive examples.		

marking where we determine the mark (tag) of a node. Kashima and Koyanagi (2002) dealt with this task by inserting the node representing the mark above the node to be tagged and classifying the transformed tree using SVMs with tree kernels such as K_{lo}. For the SRL task, Moschitti (2004) applied the tree kernel (K_c) to tree fragments that are heuristically extracted to reflect the role of interest. For relation extraction, Culotta and Sorensen (2004) proposed a tree kernel that operates on only the smallest tree fragment including two entities for which a relation is assigned. Our kernels on marked labeled ordered trees differ in what subtrees are permitted. Although comparisons are needed, we think our kernels are intuitive and general.

There are many possible structures for which tree kernels can be defined. Shen et al. (2003) proposed a tree kernel for LTAG derivation trees to focus only on linguistically meaningful structures. Culotta and Sorensen (2004) proposed a tree kernel for dependency trees. An important future task is to find suitable structures for each task (the SRL task in our case). Our speed-up method will be beneficial as long as there is unbalanced similarity.

7 Conclusion

We have presented a method for speeding up the training with tree kernels. Using the SRL task, we demonstrated that our speed-up method made the training substantially faster.

References

T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. 2002. Efficient substructure discovery from large semi-structured data. In *SIAM SDM'02*.

X. Carreras and L. Màrquez. 2005. Introduction to the CoNLL-2005 shared task: Semantic role labeling. In *CoNLL 2005*.

Y. Chi, Y. Yang, Y. Xia, and R. R. Muntz. 2004. CMTreeMiner: Mining both closed and maximal frequent subtrees. In *PAKDD 2004*.

M. Collins and N. Duffy. 2001. Convolution kernels for natural language. In *NIPS 2001*.

A. Culotta and J. Sorensen. 2004. Dependency tree kernels for relation extraction. In *ACL 2004*.

C. Cumby and D. Roth. 2003. On kernel methods for relational learning. In *ICML 2003*.

D. Gildea and D. Jurafsky. 2002. Automatic labeling of semantic roles. *Computational Linguistics*, 28(3).

K. Hacioglu, S. Pradhan, W. Ward, J. H. Martin, and D. Jurafsky. 2004. Semantic role labeling by tagging syntactic chunks. In *CoNLL 2004*.

H. Kashima and T. Koyanagi. 2002. Kernels for semi-structured data. In *ICML 2002*, pages 291–298.

T. Kudo and Y. Matsumoto. 2003a. Fast methods for kernel-based text analysis. In *ACL 2003*.

T. Kudo and Y. Matsumoto. 2003b. Subtree-based Markov random fields and its application to natural language analysis (in Japanese). *IPSJ, NL-157*.

T. Kudo and Y. Matsumoto. 2004. A boosting algorithm for classification of semi-structured text. In *EMNLP 2004*, pages 301–308.

A. Moschitti. 2004. A study on convolution kernels for shallow semantic parsing. In *ACL 2004*.

L. Shen, A. Sarkar, and A. K. Joshi. 2003. Using LTAG based features in parse reranking. In *EMNLP 2003*.

J. Suzuki, H. Isozaki, and E. Maeda. 2004. Convolution kernels with feature selection for natural language processing tasks. In *ACL 2004*, pages 119–126.

V. Vapnik. 1995. *The Nature of Statistical Learning Theory*. Springer Verlag.

S. V. N. Vishwanathan and A. J. Smola. 2004. Fast kernels for string and tree matching. *Kernels and Bioinformatics*.