Fast On-line Kernel Learning for Trees

Fabio Aiolli, Giovanni Da San Martino, Alessandro Sperduti
Dipartimento di Matematica Pura ed Applicata
Università di Padova
{aiolli,dasan,sperduti}@math.unipd.it

Alessandro Moschitti
Dipartimento di Informatica
Università di Roma "Tor Vergata"
moschitti@info.uniroma2.it

Abstract

Kernel methods have been shown to be very effective for applications requiring the modeling of structured objects. However kernels for structures usually are too computational demanding to be applied to complex learning algorithms, e.g. Support Vector Machines. Consequently, in order to apply kernels to large amount of structured data, we need fast on-line algorithms along with an efficiency optimization of kernel-based computations.

In this paper, we optimize this computation by representing set of trees by minimal Direct Acyclic Graphs (DAGs) allowing us i) to reduce the storage requirements and ii) to speed up the evaluation on large number of trees as it can be done 'one-shot' by computing kernels over DAGs. The experiments on predicate argument subtrees from PropBank data show that substantial computational savings can be obtained for the perceptron algorithm.

1. Introduction

The direct treatment of structured objects for learning and data mining applications is gaining increasing importance. Many and different application fields involve the processing of structured or semi-structured objects. For example, proteins and phylogenetic trees in Bioinformatics; hypertextual and XML documents in Information Retrieval; parse trees in Natural Language Processing. In all these areas, the huge amount of available data jointly with a poor understanding of the processes of its generation, suggest the use of machine learning and/or data mining techniques.

The major complexity in applying machine learning algorithms to structured data is the design of effective features for its representation. Kernel methods seem a valid approach to alleviate such complexity since they provide an implicit object representation and the possibility to work in very large feature spaces. Such interesting properties have triggered several researches on kernel methods for structured data, e.g., Tree kernels for NLP [2], Fisher kernels proposed in [7], convolution kernels for discrete structures introduced in [5], kernels for strings [14], kernels for Bioinformatics, e.g. [10] and so on.

In the field of Data Mining, a special attention has been devoted to find frequent trees. For example, the problem of extracting patterns in massive databases representing complex interactions among entities, usually known as the Frequent Structure Mining (FSM) task, has been addressed with rooted (ordered/unordered) labeled trees, e.g. [1, 13].

One drawback of tree kernels (in general of graph kernels) is the time complexity required by both learning and classification phases. Such complexity is typically higher than the methods based on explicit feature vectors and sometimes it prevents the kernel application in scenarios involving large amount of data. A typical approach to make treatable such complexity is the use of on-line learning/classification algorithms, e.g. the perceptron.

In this paper, we show that, when substructures are shared among the training instances (which is typically the case) we can provide a compact representation by means of Direct Acyclic Graphs (DAGs), making the computation and the storage requirement far more favorable.

We tested the benefit of our algorithm on an interesting Natural Language Processing task, namely, Semantic Role Labeling [4]. This is a text mining application, which, given the parse tree of a natural language sentence, extracts all predicates along with their arguments. A very large corpus of predicate argument structures associated with syntactic trees has been made available by the PropBank project [8]. Thus, we were able to carry out experiments with our kernel algorithm and hundreds of thousands of instances. The results show that our approach remarkably reduces both learning time and storage needs, making practical the use of tree kernels for applications in real scenarios.

2. On-line Learning and Tree Forests

In *on-line* learning, as opposed to *batch* learning, data arrives sequentially while learning takes place. Many algorithms exist tailored to this setting, the most popular of

which being the perceptron algorithm. In the original formulation the perceptron is meant to treat data constituted by real valued vectors and its decision function is linear (a hyperplane). It is well known that this algorithm can be easily extended to generate a non-linear decision function and/or to treat structured data by using kernels (see e.g. [9]).

Although the same principles can trivially be adopted to different tree-kernels, we only focus on SubSet Tree kernel (SST) defined by $K(T_1, T_2) = \sum_{t_1 \in N_{T_1}} \sum_{t_2 \in N_{T_2}} C(t_1, t_2)$, where N_{T_1} and N_{T_2} are the sets of nodes of trees T_1 and T_2, respectively, and $C(t_1, t_2)$ is recursively computed according to the rules: *i*) if $t_1 \neq t_2$ then $C(t_1, t_2) = 0$; *ii*) if $t_1 = t_2$ and t_1 is a preterminal then $C(t_1, t_2) = 1$; *iii*) if $t_1 = t_2$ and t_1 is not a preterminal then $C(t_1, t_2) = \prod_{j=1}^{nc(t_1)} (1 + C(ch_j[t_1], ch_j[t_2]))$, where $nc(t_1)$ is the number of children of t_1 and $ch_j[t]$ is the j-th child of node t. See [2] for a detailed description of SST.

The kernel-perceptron algorithm, adapted to tree-kernels, can be described as follows: the current input tree is added to the model M (initially set to the empty set) whenever its score $S(T_i) = \sum_{(T_j, y_j) \in M} y_j K(T_j, T_i)$ has different sign from its classification y_i (i.e. if T_i is misclassified):

$$\textbf{if } (y_i S(T_i) \leq 0) \textbf{ then } M \leftarrow M \cup \{(T_i, y_i)\}$$

It is trivial to show that (a) M, and consequently the memory required for its storage, grows up linearly with the number of tree presentations, and (b) the computational complexity of the function $S(T)$ is super-linear. Clearly, point (b) is not satisfactory for on-line applications.

In the next section we will show that remarkably memory savings can be obtained using a DAG for representing M.

3 The DAG-Perceptron

Computing the score function in the perceptron algorithm involves the evaluation of each kernel between the input tree and the forest of trees composing M. This computation, however, can be eased in the case in which trees belonging to the forest share common subtrees (see Figure 1). The addition of node annotations concerning the frequency of shared subtrees is sufficient to maintain all the information to re-construct the original forest. Specifically, given a tree forest F, if there are trees $T_1, T_2 \in F$ which share a common subtree $\hat{T}$, then we can explicitly represent $\hat{T}$ only once. Thus, we define a procedure that merges all the trees in F into a single *minimal* DAG, i.e., a DAG with a *minimal* number of vertices. We will refer to this DAG as $\mu D = \mu DAG(F)$. In fact, this procedure will produce an annotated DAG (ADAG), i.e. a DAG where each node is annotated with a pair (*label*,*frequency*). The *label* represents information associated with the node, while the *frequency* is used to count how many repetitions of the same subtree rooted in that node are present in the tree forest. The exact use of this last field will become clearer in the following.

In Figure 2, we give an algorithm to efficiently compute shared subtrees, and exploit this information to efficiently represent a forest as an ADAG. The procedure `InvTopologOrder`(T_j) used in step 3 returns a total order of vertexes of T_j which is compatible with the (inverted) partial order defined by the arcs of T_j. Thus, the first vertexes of the list will be vertexes with zero outdegree (leaves), followed by vertexes which have only children with zero outdegree, and so on. Using this order guarantees the (unique) existence of vertexes $c_i \in \mu D$ s.t. $dag\_rooted(c_i) \equiv dag\_rooted(ch_i[v])$ in step 11. In fact, for each i, the vertex $ch_i[v]$ is processed before vertex v is either inserted in μD at step 9 or recognized as a duplicated of a vertex already present in μD at step 6.

It should be noted that the function $dag\_rooted(\cdot)$ can be implemented quite efficiently by an indexing mechanism, where a unique code is defined for a void child, and a unique code for the root of each different DAG is generated by recursively considering the label of the root and the (unique) codes computed for its children. In our implementation we have realized an indexing mechanism by using AVL trees. Let t be a vertex of a tree T and l the length of the longest path in T starting from t and reaching a vertex of T with 0 outdegree. Then an AVL tree for each possible value of l is defined, i.e. $AVL^{(l)}$. When a vertex $s \in T$ with 0 outdegree is processed, there is an attempt to insert it in $AVL^{(0)}$ using as key the label associated with s. If the key is already present, it means that a vertex s' with 0 outdegree and same label has already been inserted in $AVL^{(0)}$. In that case, s is marked, the frequency for s' is incremented by 1, and the pointer to s' is associated with it, so that, when the parents of s are processed, their pointers to s are substituted by the pointer to s'. When all the vertexes with 0 outdegree are processed, vertexes with $l = 1$ are considered and the same process is repeated with the following two differences: *i)* the children of q are checked and for each marked child, its pointer is substituted by the associated pointer; *ii)* the key used for the insertion in $AVL^{(1)}$ is given by the con-

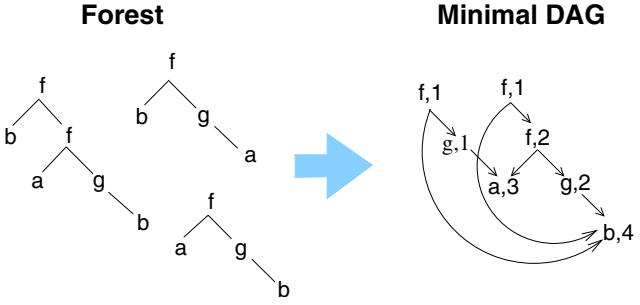

Figure 1. Example of Forest optimization.

MinimalDAG

Input: A tree forest $F = \{T_1, \ldots, T_k\}$ 0

/* $l \equiv label$, $f \equiv frequency$, $dag \equiv dag\_rooted$ */

Initialize: $\mu D \leftarrow$ void DAG; 1

for $j \leftarrow 1$ **to** N **do** 2

$vertex\_list \leftarrow$ `InvTopologOrder`(T_j); 3

while $vertex\_list \neq \emptyset$ **do** 4

$v = pop(vertex\_list)$; 5

if $\exists u \in \mu D$ s.t. $dag(u) \equiv dag(v)$ 6

then $f(u) \leftarrow f(u) + f(v)$ 7

else 8

add to μD a node w where 9

$l(w) = l(v)$ and $f(w) = f(v)$

forall children $ch_i[v]$ of v 10

add arc (w, c_i) to μD where 11

$c_i \in Nodes(\mu D)$ and

$dag(c_i) \equiv dag(ch_i[v])$

return μD 12

Figure 2. The algorithm to transform a tree-forest into a minimal (annotated) DAG.

DAGIns

Input: An ADAG μD and a weighted annotated DAG (D, α) to be inserted 0

/* $l \equiv label$, $f \equiv frequency$, $dag \equiv dag\_rooted$ */

$vertex\_list \leftarrow$ `InvTopologOrder`(D); 1

while $vertex\_list \neq \emptyset$ **do** 2

$v = pop(vertex\_list)$; 3

if $\exists u \in \mu D$ s.t. $dag(u) \equiv dag(v)$ 4

then $f(u) \leftarrow f(u) + \alpha \cdot f(v)$ 5

else 6

add to μD a node w where 7

$l(w) = l(v)$ and $f(w) = \alpha \cdot f(v)$

forall children $ch_i[v]$ of v 8

add arc (w, c_i) to μD where 9

$c_i \in Nodes(\mu D)$ and

$dag(c_i) \equiv dag(ch_i[v])$

return μD 10

Figure 3. The algorithm to insert a weighted ADAG in a larger ADAG.

catenation of the label associated with q with the ordered sequence of (revised) pointers to its children. If the insertion of q fails, i.e., an "equivalent" vertex is already present, the same operations described for s are executed. The treatment of vertexes with $l > 1$ is the same described for the case $l = 1$. Both insertion and lookup into an AVL tree take $O(\log(n))$, where n is the number of items contained into the AVL tree[1]. A better implementation, on average, can be obtained by substituting AVL trees with corresponding hash tables, which have average complexity $O(1)$ for lookup. Potential problems for hash tables are a worst case of $O(n)$ for lookup and the time spent to evaluate the hash function, which can constitute a significant overhead.

The idea of our approach is to use the ADAG in place of the original tree forest in order to make the computation of the perceptron score more efficient. For this, it is useful to notice that the core of the kernel computation is the computation of the $C(t_i, t_j)$'s which can be computed just once for the shared substructures and re-used when needed.

In general, the score function is given in the form

$$S(T) = \sum_{T_i \in F} \alpha_i K(T_i, T)$$

[1]Notice that using a different AVL tree for each value of l allows us to reduce the number of vertexes inserted in the AVL, thus reducing the searching time for the key.

where $\alpha_i \in \mathbb{R}$ are weights. This can be efficiently computed by keeping in memory an ADAG (the model) incrementally built during learning. When adding a new tree T_i, the frequencies of the model nodes are simply updated with the frequency of the nodes of the μDAG of T_i (weighted by α_i) In the special case of perceptron, $\alpha_i = y_i$. The algorithm used for inserting a new ADAG into the model is depicted in Figure 3. Note that it is very similar to the generation of a minimum DAG with the difference being that in this case the frequencies are updated with weights α.

The following theorem shows that the weighted subtree frequencies, maintained in the model as a minimal ADAG, allow us to compute the score $S(T)$ without making explicit reference to the trees in the standard perceptron model.

Theorem: Let $M_0 = \phi$ the void initial DAG. After n insertions $M_i = DAGIns(M_{i-1}, \mu DAG(T_i), \alpha_i)$, where $i = 1, \ldots, n$. Defining

$$S_{\mu DAG}(M_n, \mu DAG(T)) = \sum_{t_i \in M_n} \sum_{t_k \in \mu DAG(T)} f_i \bar{f}_k C(t_i, t_k),$$

where f_i and $\bar{f}_k$ are the weighted frequencies in M_n and $\mu DAG(T)$, respectively, then the following holds:

$$S(T) = S_{\mu DAG}(M_n, \mu DAG(T)).$$

Proof: Let us consider the set of all the possible subtrees S_k indexed by $k = 1, ..., m_S$. First of all, we can check easily

that, if the algorithm in Figure 3 is used to insert n trees into the model, starting from the void model, then we have:

$$f_k = \sum_{i=1}^{n} \alpha_i c_k(T_i) \quad (1)$$

where $c_k(T)$ is the number of times a given subtree S_k appears in a tree T.

Now, let $root(S)$ be the root node of a tree S, we have

$$\begin{aligned} K(T_i, T) &= \sum_{t_i \in T_i} \sum_{t \in T} C(t_i, t) \\ &= \sum_{k,j} c_k(T_i) c_j(T) C(root(S_k), root(S_j)) \end{aligned}$$

where k and j vary over the space of all possible subtrees. The equality above is true because $c_i(T) = 0$ whenever the subtree S_i is not present in T. Hence,

$$\begin{aligned} S(T) &= \sum_{i=1}^{n} \alpha_i K(T_i, T) \\ &= \sum_{i=1}^{n} \alpha_i \sum_{k,j} c_k(T_i) c_j(T) C(root(S_k), root(S_j)) \\ &= \sum_{k,j} f_k \bar{f}_j C(root(S_k), root(S_j)) \end{aligned}$$

This last equality is true because of eq. 1 and because in a minimal DAG $\bar{f}_j = c_j(T)$ is true by definition.

Now, since $f_k = 0$ when the subtree S_k is not a subgraph of M_n and $\bar{f}_j = 0$ when the subtree S_j is not a subgraph of $\mu DAG(T)$, then we obtain

$$\begin{aligned} S(T) &= \sum_{t_k \in M_n} \sum_{t_j \in \mu DAG(T)} f_k \bar{f}_j C(t_k, t_j) \\ &= S_{\mu DAG}(M_n, \mu DAG(T)) \qquad \square \end{aligned}$$

Using this more general result, it is not difficult to define an implementation of the Perceptron algorithm where the model, i.e. the forest of misclassified trees with their labels, is maintained as an ADAG. The algorithm can be summarized as follows: the current input tree is added to the model M (initially set to a void DAG) if its score $S(T_i) = S_{\mu DAG}(M, \mu DAG(T_i))$ has different sign from its classification y_i, i.e.:

if $(y_i S(T_i) \leq 0)$ **then** $M \leftarrow DAGIns(M, (\mu DAG(T_i), y_i))$

This time the model is represented as a minimal ADAG and updated at each error by the insertion of the minimal ADAG obtained by the input tree that caused the error.

The soundness of the algorithm is guaranteed by the theorem given in the previous section which is however valid for a larger class of algorithms.

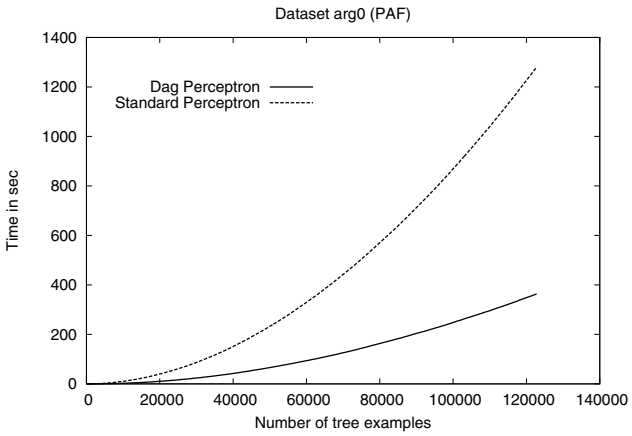

Figure 4. Comparison of the training times of the standard perceptron versus the dag-based implementation for PAF (Arg0).

4 Experiments

The DAG and standard perceptron were compared with respect to computational time and memory requirement. For this purpose, we used the PropBank dataset (`www.cis.upenn.edu/~ace`) along with PennTree bank 2. This large corpus contains about 53,700 sentences that we split in training set (sections from 02 to 21) and test set (Section 23), for a total of 122,774 and 7,359 predicate arguments (from Arg0 to Arg5, ArgA and ArgM).

Following the approach in [11], we reduced the problem of classifying arguments to the one of classifying subtrees encoding the predicate/argument relation. In particular, we used two different linguistic objects: (a) the minimal subtree that includes a predicate with only one of its arguments (PAF) and the subtree which includes the subcategorization frame of the target verbal predicate (SCF). According to [11], the above objects can be directly plugged in a tree-kernel based machines to learn the classification function.

The experiments were carried out with our implementation of the models proposed in sections 2 and 3. In particular, we focused on the classification of PAF (Arg0), SCF (Arg0), and SCF (Arg1). Space limits force us to report detailed results for PAF (Arg0) only. However, results for the other datasets are qualitatively similar.

Figure 4 reports the plots of the time required to train the two perceptron algorithms. The number of trees is reported on the x-axis, while the corresponding CPU[2] time in seconds is reported on the y-axis. The number of trees inserted in the model at the end of learning were $10,266$ PAFs describing Arg0 and $9,926$ and $23,781$ SCFs representing

[2]The computer used for the experiments is based on an AMD Athlon(tm) 64 Processor 3500+.

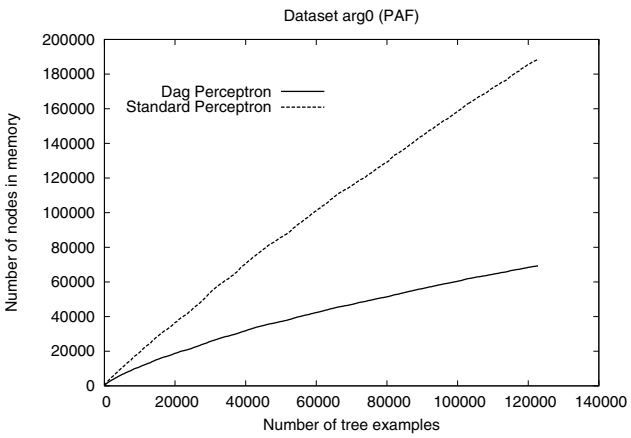

Figure 5. Comparison of the number of vertexes inserted in the model by the standard perceptron versus the dag-based implementation for PAF (Arg0) as a function of the number of input trees.

Arg0 and Arg1, respectively. The final speed up (according to different classifiers and datasets) ranges, from 3.52 when PAF trees for Arg0 are used, to 5.45 with SCF trees, up to 6.85 when SCF trees of Arg1 are used.

Figure 5 reports the plots of the number of vertexes inserted in the model by the standard and dag-based implementations according to the PAF (Arg0) dataset. As expected, the number of vertexes inserted by the dag-based implementation is much less and this is the main reason for the speed up in computation.

5 Conclusions

On-line learning is important when huge amount of data have to be processed. State of the art learning techniques suggest the use of kernels which are computational demanding when dealing with structured objects. This paper shows that, when considering tree kernels it is actually possible to reduce the computational burden and storage requirement by representing common subtrees with annotated minimal DAGs. In particular, we have shown that substantial computational savings can be obtained for the perceptron algorithm using the SST kernel over a quite extensive dataset made available by the PropBank project.

It is important to stress that the same basic idea can be exploited in all the learning algorithms where the decision function is computed as a linear combination of kernel evaluations, such as perceptron with margin, Support Vector Machines [3], boosting [12] and bayes point machines [6].

As a final remark we observe that the concepts introduced in this paper allow to define a kernel function between annotated minimal DAGs which can then be used to define a non trivial kernel for standard DAGs. In fact, given two annotated DAGs, D_1, D_2, with frequencies f_k^1, f_j^2 associated with their nodes, we can define a similarity score between them by using the formula:

$$\tilde{K}(D_1, D_2) = \sum_{\substack{t_k \in D_1 \\ t_j \in D_2}} f_k \bar{f}_j C(t_k, t_j)$$

It should be noticed that $\tilde{K}$ is a valid kernel (positive definite kernel) whenever both the annotated DAGs are constructed as minimal DAGs of tree forests, i.e. $\exists F_1, F_2$ s.t. $D_1 = \mu DAG(F_1)$, $D_2 = \mu DAG(F_2)$.

References

[1] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *SDM*, 2002.

[2] M. Collins and N. Duffy. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *ACL02*, 2002.

[3] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[4] D. Gildea and D. Jurasfky. Automatic labeling of semantic roles. *Computational Linguistic*, 28(3):496–530, 2002.

[5] D. Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California, Santa Cruz, July 1999.

[6] R. Herbrich, T. Graepel, and C. Campbell. Bayes point machines. *Journal of Machine Learning Research*, 1:245–279, 2001.

[7] T. Jaakkola, M. Diekhans, and D. Haussler. A discriminative framework for detecting remote protein homologies. *Journal of Computational Biology*, 7(1,2):95–114, 2000.

[8] P. Kingsbury and M. Palmer. From Treebank to PropBank. In *Proceedings of LREC'02)*, Las Palmas, Spain, 2002.

[9] J. Kivinen, A. J. Smola, and R. C. Williamson. Online learning with kernels. *Signal Processing, IEEE Transactions on*, 52(8):2165–2176, 2004.

[10] R. Kuang, E. Ie, K. Wang, K. Wang, M. Siddiqi, Y. Freund, and C. S. Leslie. Profile-based string kernels for remote homology detection and motif extraction. In *3rd International IEEE Computer Society Computational Systems Bioinformatics Conference (CSB 2004)*, pages 152–160, 2004.

[11] A. Moschitti. A study on convolution kernel for shallow semantic parsing. In *Proceedings of ACL'04*, Barcelona, Spain, 2004.

[12] R. E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[13] A. Termier, M.-C. Rousset, and M. Sebag. Dryade: A new approach for discovering closed frequent trees in heterogeneous tree databases. In *ICDM*, pages 543–546, 2004.

[14] C. Watkins. Dynamic alignment kernels. Technical Report CSD-TR-98-11, Royal Holloway, University of London, January 1999.