Efficient Kernel-based 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 are effective approaches to the modeling of structured objects in learning algorithms. Their major drawback is the typically high computational complexity of kernel functions. This prevents the application of computational demanding algorithms, e.g. Support Vector Machines, on large datasets. Consequently, on-line learning approaches are required. Moreover, to facilitate the application of kernel methods on structured data, additional efficiency optimization should be carried out.**

In this paper, we propose Direct Acyclic Graphs to reduce the computational burden and storage requirements by representing common structures and feature vectors. We show the benefit of our approach for the perceptron algorithm using tree and polynomial kernels. The experiments on a quite extensive dataset of about one million of instances show that our model makes the use of kernels for trees practical. From the accuracy point of view, the possibility of using large amount of data has allowed us to reach the state-of-the-art on the automatic detection of Semantic Role Labeling as defined in the Conference on Natural Language Learning shared task.

I. Introduction

Many and different data mining applications involve the processing of structured or semi-structured objects, e.g. proteins and phylogenetic trees in Bioinformatics, molecular graphs in Chemistry, 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 for its generation, typically enforces the use of machine learning and/or data mining techniques.

Many researchers in data mining field have focused on the task of finding 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. [3], [23].

The major complexity of 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 allow to inject background knowledge into a learning algorithm and provide an implicit object representation with the possibility to work in very large feature spaces. Such interesting properties have triggered several researches on kernel methods for structured data, e.g., Fisher kernels proposed in [10], convolution kernels for discrete structures introduced in [8], kernels for strings [24], kernels for Bioinformatics [13], [14], [20] and so on.

One drawback of tree kernels (and kernels for structures in general) is the time complexity required both in learning and classification phases. Such complexity can sometimes prevent the kernel application in scenarios involving large amount of data. Two approaches have been pursued in order to allow the use of kernels in data mining: (1) the use of fast learning/classification algorithms, e.g. the perceptron; (2) the reduction of the computational time required for computing a kernel (see e.g. [22] and references therein).

Methods based on the maximization of the margin are the state-of-the-art for classification algorithms and are supported by a solid theory. However, it is well-known that the availability of very large collections of labeled examples increases the generalization performance of a given algorithm. Such increase fills the gap between the expected generalization performance of simple on-line algorithms, which are not designed to maximize the margin, and algorithms which explicitly maximize it, as Support Vector Machines (SVMs) for example. As a consequence, algorithms able to work with very large collections of examples can outperform state-of-the-art algorithms which cannot deal with such a number of examples.

In a previous paper [2], we have shown that, when substructures are shared among the training instances (and this is usually the case for discrete structured objects), we can provide a compact representation of the structures by means of Direct Acyclic Graphs (DAGs).

Exploiting this idea has led to a significant reduction both in model storage and kernel computational time requirements. This idea has been applied to the case of the perceptron algorithm, obtaining the so called DAG-Perceptron [2].

However, in [2], we focused on efficient representation of structures, whereas in this paper, we introduce novel optimizations which significantly reduce memory requirements during computation of kernels. Moreover, we modified DAG algorithm to compute generic kernels on feature vectors extracted from structured data. In particular, we consider standard features which are attached to trees. In the original version of the DAG-Perceptron, we combined the linear kernel over such features with tree kernels. In this paper, we also give an algorithm for the efficient computation of non-linear kernels for sparse feature vectors, with the aim to

boost the overall accuracy.

To demonstrate the significance of this improvement, we tested our algorithm against an interesting Natural Language Processing task, namely Semantic Role Labeling (SRL) [7]. Given the parse tree of a natural language sentence, an SRL system 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 [11]. The results of our experiments with kernel algorithms on millions of items show that our approach remarkably reduces both learning time and storage needs. Moreover, on the SRL Conference Natural Language Learning (CoNLL) 2005 shared task dataset, we approached the highest accuracy in detecting arguments. This is due to the fact that for the first time tree kernels (along with traditional feature vectors) could be used on such huge dataset. In other words, our approach makes the use of tree kernels for applications practical on real scenarios, where running more complex algorithm, e.g. SVMs, would just have resulted prohibitive.

Sections II and III introduce tree kernels and the perceptron algorithm, respectively. Section IV describes the basic idea of the DAG-based algorithm, while in Section IV-C our proposed algorithm together with further optimization are described. Section VI empirically shows the benefits of our approach and finally, Section VII summarizes the conclusions.

II. Kernels for Trees

One of the most used tree kernel for Natural Language processing is the Subset Tree (SST) kernel [5]. It is based on counting matching subset trees between two input trees. A subset tree (SST) of a tree T is a tree which (a) is rooted in a node of T and (b) satisfies the constraint that each of its nodes contains either all its children or none of them. Note that leaves do not need to be necessarily included in the SST.

We assume that each of the m SSTs in the whole training data set is indexed by an integer between 1 and m. Then $h_s(T)$ is the number of times the ***subset*** tree indexed with s occurs in ***tree*** T. We represent each tree T as a feature vector $\phi(T) = [h_1(T), h_2(T), \dots]$. The inner product between two trees under the representation $\phi(T) = [h_1(T), h_2(T), \dots\ h_m(T)]$ is:

$$K(T_1, T_2) = \phi(T_1) \cdot \phi(T_2) = \sum_{s=1}^{m} h_s(T_1) h_s(T_2).$$

Thus this tree kernel defines a similarity measure between trees which is proportional to the number of shared SSTs.

The SST can be efficiently calculated by a recursive procedure defined as follows:

$$\begin{aligned} K(T_1, T_2) &= \sum_{t_i \in N_{T_1}} \sum_{t_2 \in N_{T_2}} \sum_{s=1}^{m} h_s(t_1) h_s(t_2) \quad (1) \\ &= \sum_{t_1 \in N_{T_1}} \sum_{t_2 \in N_{T_2}} C(t_1, t_2) \end{aligned}$$

where N_{T_1} and N_{T_2} are the sets of nodes of trees T_1 and T_2, respectively, and $C(t_1, t_2) = \sum_{s=1}^{m} h_s(t_1) h_s(t_2)$. Let a *production at* node t be the SST constituted by t and only its direct children, then $C(t_1, t_2)$ can be recursively computed according to the following rules:

1) if the productions at t_1 and t_2 are different then $C(t_1, t_2) = 0$;
2) if the productions at t_1 and t_2 are the same, and t_1 and t_2 have only leaf children (i.e. they are pre-terminals symbols) then $C(t_1, t_2) = 1$;
3) if the productions at t_1 and t_2 are the same, and t_1 and t_2 are not pre-terminals, then

$$C(t_1, t_2) = \prod_{j=1}^{nc(t_1)} (1 + C(ch_j[t_1], ch_j[t_2])) \quad (2)$$

where $nc(t_1)$ is the number of children of t_1 and $ch_j[t]$ is the j-***th*** child of node t.

The computational complexity in time of the above kernel is $O(|N_{T_1}| \times |N_{T_2}|)$. An improvement[1] to the procedure can be introduced by observing that when the productions associated with t_1 and t_2 are different, we can avoid to compute $C(t_1, t_2)$ since it is 0. The resulting Fast Tree Kernel algorithm (FTK) [17] has the same worst case complexity but in practical applications it provides a quite relevant speed-up.

III. On-line Learning, Perceptron, and Tree Forests

In *on-line* learning, as opposed to *batch* learning, data arrives sequentially while learning takes place. Many algorithms tailored to this setting exist, the most popular being the perceptron algorithm [19]. In the original formulation the perceptron was meant to process data encoded by real vectors and the 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 for example [12]).

The on-line kernel-perceptron algorithm, adapted to tree-kernels, requires to maintain an implicit representation of the weight vector in the feature space. Specifically, this corresponds to keep in memory the set of the already seen examples for which the perceptron prediction was erroneous. In fact, let $\mathcal{T}$ be a stream of example pairs (T_i, y_i), $y_i \in \{-1, +1\}$, then, at iteration t, the scoring function of the perceptron for a new tree T is defined by

$$S_t(T) = \sum_{j=1}^{t-1} \alpha_j K(T_j, T_i),$$

where $\alpha_j \in \{-1, 0, +1\}$ is 0 whenever $sign(S_j(T_j)) = y_j$, and y_j otherwise.

Thus we can consider the set of trees $M = \{(T_i, y_i) \in \mathcal{T} : \alpha_i \in \{-1, +1\}\}$ as the *model* of the perceptron and slightly

[1] Another improvement from an accuracy point of view is carried out by downweighting larger subtrees. This is obtained by modifying the kernel as follows: $K(T_1, T_2) = \sum_{s=1}^{m} \lambda^{\text{size}(s)} h_s(T_1) h_s(T_2)$ where $0 < \lambda \leq 1$ is a weighting parameter and size(s) is the number of nodes of the subtree s.

redefine the kernel-perceptron algorithm as in the following. Let $M = \emptyset$ be an initial empty model, a new input tree T_i is added to the model M whenever its score

$$S(T_i) = \sum_{(T_j, y_j) \in M} y_j K(T_i, T_j)$$

has different sign from its classification y_i. Thus the update and the insertion of the new example follow the rule:

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

It is trivial to show that the cardinality of M, and consequently the memory required for its storage, grows up linearly with the number of tree presentations. Moreover the efficiency in the evaluation of the function $S(T)$ decreases super-linearly. Clearly, this seems not satisfactory for on-line applications.

In [2], it has been shown that, in the case of discrete structures as trees and DAGs, where many sub-structures are shared, we can compact the representation of the model thus making the computation faster and reducing the storage requirement remarkably.

IV. The DAG-Perceptron

The perceptron algorithm maintains a tree forest including all the trees that have been misclassified until that moment. The computational burden is thus concentrated in computing the score which involves the computation of kernels between the input tree and each tree of the forest. This computation, however, can be eased when 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.

This section is organized as follows: subsections IV-A and IV-B recall the optimizations described in [2], while subsection IV-C discusses further optimizations carried out in this paper.

A. From a Forest to a Directed Acyclic Graph

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 Figure 2, we give an algorithm to efficiently compute shared subtrees, and how to exploit this information to efficiently represent a forest as an annotated DAG (ADAG). More formally, with annotated DAG, we refer to a DAG where each node is annotated with a pair (*label*,*frequency*). *label* represents the structure associated with the node, while *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 *frequency* will become clearer in the following.

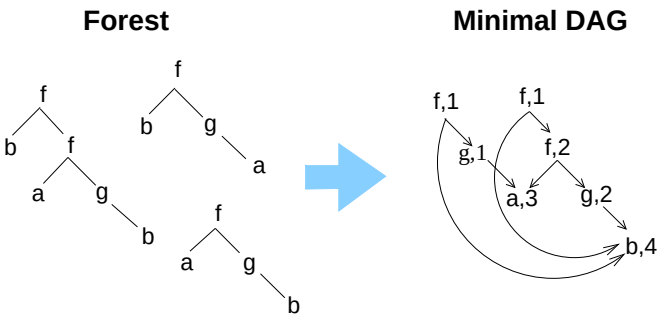

Fig. 1. Example of Forest optimization.

MinimalDAG

Input: A tree forest $F = \{T_1, \ldots, T_k\}$
/* $l \equiv label$, $f \equiv frequency$, $dag \equiv dag\_rooted\_at$ */
Initialize: $\mu D \leftarrow$ void DAG;
  for $j \leftarrow 1$ **to** N **do**
  $vertex\_list \leftarrow$ `InvTopologOrder`(T_j);
  while $vertex\_list \neq \emptyset$ **do**
    $v = pop(vertex\_list)$;
    if $\exists u \in \mu D$ s.t. $dag(u) \equiv dag(v)$
      then $f(u) \leftarrow f(u) + f(v)$
      else
        add to μD a node w where
          $l(w) = l(v)$ and $f(w) = f(v)$
        forall children $ch_i[v]$ of v
          add arc (w, c_i) to μD where
            $c_i \in Nodes(\mu D)$ and
            $dag(c_i) \equiv dag(ch_i[v])$
  return μD

Fig. 2. The algorithm to transform a tree-forest into a minimal DAG.

The procedure `InvTopologOrder`(T_j) used in the algorithm 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, 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. $dag\_rooted\_at(c_i) \equiv dag\_rooted\_at(ch_i[v])$. In fact, for each i, the vertex $ch_i[v]$ is processed before vertex v and is either inserted in μD or recognized as a duplicated of a vertex already present in μD.

It should be noted that the function $dag\_rooted\_at(\cdot)$ 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 Adelson-Velsky Landis (AVL) trees [1]. 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 concatenation 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.

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.

The advantage of having a minimal DAG leads to a considerable reduction in space complexity. In some cases, such reduction can even be exponential.

TreeIns

Input: An ADAG μD and a tree (T, α) to be inserted
/* $l \equiv label$, $f \equiv frequency$, $dag \equiv dag\_rooted\_at$ */
$vertex\_list \leftarrow$ `InvTopologOrder`(T);
while $vertex\_list \neq \emptyset$ **do**
$v = pop(vertex\_list)$;
if $\exists u \in \mu D$ s.t. $dag(u) \equiv dag(v)$
then $f(u) \leftarrow f(u) + \alpha \cdot f(v)$
else
add to μD a node w where
$l(w) = l(v)$ and $f(w) = \alpha \cdot f(v)$
forall children $ch_i[v]$ of v
add arc (w, c_i) to μD where
$c_i \in Nodes(\mu D)$ and
$dag(c_i) \equiv dag(ch_i[v])$
return μD

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

B. Efficient Score Computation

In the previous section we showed how to transform a tree forest into an annotated DAG with no loss of information. In this section we show a way to compact the necessary information to be used in the computation of a pretty general score function.

The general idea of our approach is to compute an annotated minimum DAG for the original tree forest in a way that it will be possible to perform the computation of the perceptron score function efficiently. For this, it is useful to notice that the core of the computation of kernels is the computation of the $C(t_i, t_j)$ and that these values can be computed just once for the shared substructures and re-used when needed.

The score function is generally given in the form

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

where $\alpha_i \in \mathbb{R}$ are weights. This can be efficiently computed by keeping in memory a weighted annotated DAG which is built incrementally during the learning and where the frequencies are computed by weighting the frequency of the minimal DAG associated with the tree which is added to the model.

The algorithm used to insert a new tree 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 frequency associated with the model is updated with the frequency of the subtree to ***be added*** weighted by the quantity α.

The following theorem shows that the weighted subtree frequencies maintained in the model, as an annotated minimal DAG, 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 $M_i = DAGIns(M_{i-1}, (T_i, \alpha_i))$, where $i = 1, \ldots, n$. Defining

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

with f_i the weighted frequency in M_n, then the following holds:

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

Proof: The proof of this theorem trivially follows from the theorem presented in [2] and is not reported due to space limitations.

C. The Algorithm

In this section, we describe an implementation of the Perceptron algorithm where the model, i.e. the forest of misclassified trees with their labels, is maintained as an annotated DAG.

The algorithm is presented in Figure 4. The model is represented as a annotated minimal DAG. Whenever an input tree is misclassified the model is updated by adding it to the model. In doing that, the weight of each node of the input tree is set to the target label (i.e. $\alpha_i = y_i$).

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.

In the following, we discuss other complexity issues which were not considered in [2], but are important when dealing with a large amount of data.

The computation of the tree kernel is based on the recursive computation of the $C(t_1, t_2)$ values. When considering

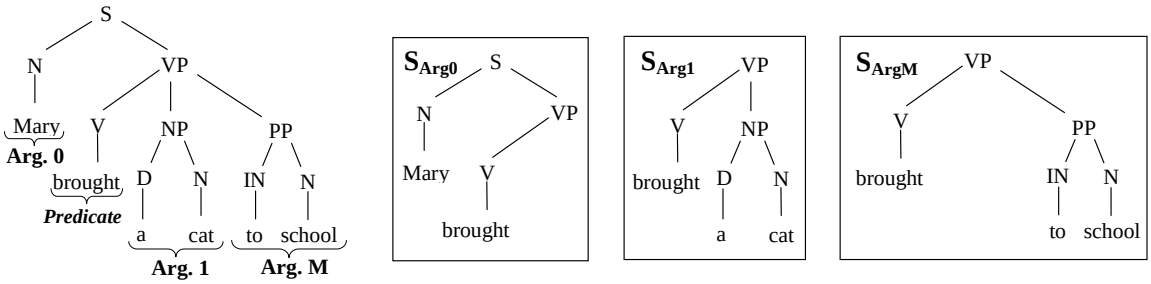

Fig. 5. Parse tree of the sentence "Mary brought a cat to school" along with the PAF trees for Arg0, Arg1 and ArgM.

DAG-Perceptron Algorithm

Input: stream of pairs (T_i, y_i), where $y_i \in \{-1, +1\}$
Initialize: Model $M \leftarrow$ void DAG;
Repeat forever

read (T_i, y_i) from the stream;
Compute Perceptron score:

$$S(T_i) \leftarrow S_{\mu DAG}(M, T_i);$$

if $y_i S(T_i) \leq 0$ **then**

$$M \leftarrow TreeIns(M, (T_i, y_i))$$

Fig. 4. The DAG-Perceptron algorithm.

a model represented as a tree forest, storing all the $C(t_1, t_2)$ values is not a problem since the computation of the total kernel is done by summing the values of the kernels between each tree in the forest and the input tree. Since the same storage space can be reused for different trees in the forest, the storage requirement is dominated by the largest tree in the forest. In [2], the above approach is used to compute kernels between an input tree and a DAG. Since the number of DAG vertexes grows with training size a significant storage requirement is expected, especially when considering data mining applications, where the number of input items could be more than a million.

For this reason, it is important to limit storage requirements. In this respect, two observations can be done: *i)* when considering a vertex v belonging to the input tree, it is readily evident that when all the $C(u, v)$ entries, with u belonging to the DAG, are computed, the entries referring to children of v can be removed, since no other tree vertex will refer to them; *ii)* an entry $C(u, v)$ is computed (and thus stored) only if *production(u)=production(v)*, thus the "name" of a vertex u belonging to the DAG, can be defined as the composition of *production(u)* plus a progressive numerical id assigned to the vertexes of the DAG bearing the same *production(u)*. Equivalently this means that, given a vertex v in the input tree, the elements of the row $C(\cdot, v)$ can be enumerated progressively, disregarding the 0 valued elements, which correspond to vertexes in the DAG that do not bear the same production as v.

On the basis of these observations, the following joint strategies can be adopted to reduce the storage requirements: *i)* the input tree is read using a depth-first visit and as soon as a vertex completes the computation of its $C(u, v)$ entries, the storage space for the $C(u, v)$ entries referring to its children is deallocated; *ii)* for each distinct production, a list of matching vertexes in the DAG is maintained with the aim of both speeding up the search for a production match, and also to assign a progressive numerical id to the matching vertexes as well as the total number of matching vertexes to the production; in this way, when a new vertex in the input tree is visited, it is possible to know how much storage space must be dynamically allocated for that vertex. It should be noticed that each list associated with a production can be maintained very efficiently by just: (1) using a counter c recording the current total number of vertexes belonging to the list; (2) assigning as id to a new vertex the current value of c; (3) inserting the new vertex at the beginning of each list and incrementing c by 1. All the above operations can be done in constant time. Adopting the above strategies reduces the storage need from $O(N_{dag} N_{tree})$, where N_{dag} is the number of nodes in the DAG, and N_{tree} is the number of nodes in the input tree, to $O(P_{max} h_{tree} b_{tree})$, where P_{max} is the length of the longest list of matching vertexes associated with productions, h_{tree} is the depth of the input tree, and b_{tree} is the branching factor of the input tree. Of course, when considering more than 1 production, $N_{dag} > P_{max}$ and if there are q productions with the same probability to be associated with a vertex, $P_{max} = \frac{N_{dag}}{q}$. Moreover, $N_{tree} \geq h_{tree} b_{tree}$.

Another efficiency issue is related to the possibility to exploit additional numerical features $\boldsymbol{\xi} \equiv [\xi_1, \ldots, \xi_d]$ associated with each tree. In that case, the score can be obtained as a combination of the score obtained by the tree kernel with the score obtained by these numerical features. For example, if the combination is the sum, the score can be computed as

$$S(T_i) = S_{\mu DAG}(M, T_i) + S_F(M_F, \xi_i),$$

where M_F is the set of feature vectors plus labels corresponding to errors. When using a nonlinear kernel for the computation of the feature score, a proper treatment is due. In fact, let consider the generic computation of the score for the features

$$S_F(\boldsymbol{\xi}_i) = \sum_{(\boldsymbol{\xi}_j, y_j) \in M_F} y_j K(\boldsymbol{\xi}_i, \boldsymbol{\xi}_j)$$

If d is large, assuming that the computation of the kernel is $O(d)$, the computational complexity for the score is $O(d|M_F|)$.

If the $\boldsymbol{\xi}$ vectors are sparse, let say that no more that $k \ll d$ components are nonzero, then a more efficient computation can be performed. In fact, nonzero features of $\boldsymbol{\xi}$ vectors can be organized for fast access by feature id. Assuming that the probability for a feature to be nonzero is $l = \frac{k}{d}$, this means that each feature will be associated with an inverted list with expected length equal to $l|M_F|$. Thus, given an input feature, for each j a match in its inverted list should be found, which can be done in no less than $O(\log(l|M_F|))$ by exploiting the sorting of the items by vector index. This leads to a total complexity of $O(|M_F|k\log(l|M_F|))$. However, we can do better than this. In fact, we know that all the items contained into the inverted lists of matching features are used for computing the score. The only problem is to recognize for each j which are the features that match the input. This can be done using the following procedure. We assume that the inverted lists are sorted by decreasing vector index. First of all the inverted lists corresponding to matching input features are recovered and their heads are inserted into a max heap. Then the maximum value is extracted by the heap and its successor in the corresponding inverted list is inserted into the heap. This process is repeated giving origin to a stream of indexes extracted by the heap where equal indexes are clustered together, allowing the computation of the kernel for that index. This procedure has a complexity that is dominated by the insertion into the heap of all the items into the matching inverted lists. Since the heap will never contain more than k items, insertion costs $\log(k)$, while the total number of items is $kl|M_F|$. Thus the total complexity is $O(kl|M_F|\log(k))$, which is better than the previous one only if $l\log(k) < \log(l|M_F|)$, i.e., $k^l < l|M_F|$. Noticing that $l \in [0,1]$, it is not difficult to realize that when $k \ll |M_F|$ a significant savings in computation can be obtained. For example, assuming binary data structures are used, if $|M_F| = 2^{14}$, $l = 10^{-4}$, and $k = 2^5$, we have $l\log_2(k) = 0.0005$ versus $\log_2(l|M_F|) = 0.71228762$, with a speedup of more than 1424.

V. A Semantic Application of Parse Tree Kernels

One of the ultimate goals of Natural Language Processing is to automatically derive semantic information from texts. Given the complexity of such task, most of the current studies focus on shallow approaches to semantic parsing. Among others, the PropBank project [11] proposes predicate argument structures to encode shallow semantics from texts. The basic assumption is that such predicative structures are strictly connected to the syntax of the textual sentences. Figure 5 exemplifies such idea by showing the parse tree of the sentence: `"Mary brought a cat to school"` along with the predicate argument annotation proposed by the PropBank project. Only verbs are considered as predicates whereas arguments are labeled sequentially from Arg0 to Arg5 plus ArgMs including several type of adjuncts.

Previous work has shown that the automatic PropBank argument annotation, i.e. Semantic Role Labeling (SRL), can be carried out by applying machine learning techniques, e.g. [7], [18]. These latter represent predicate argument relationships with vectors of features extracted from the syntactic parse tree of the target sentence. Such *standard* features, firstly proposed in [7], refer to flat information derived from parse trees, i.e. *Phrase Type*, *Predicate Word*, *Head Word*, *Governing Category*, *Position* and *Voice*.

For example, *Phrase Type* is the label of the *argument node*, i.e. the node that dominates all and only the argument words. In Figure 5 the values of such feature are `N`, `NP` and `PP` for Arg0, Arg1 and ArgM, respectively. The *Parse Tree Path*, instead, represents the path in the parse-tree between a predicate node and one of its argument nodes. It is expressed as a sequence of nonterminal labels linked by direction symbols (up or down), e.g. `V`↑`VP`↓`NP` is the path between the predicate and Arg1.

An alternative representation proposed in [16], is based on the application of tree kernels to subtrees encoding the predicate/argument relation. More precisely, each predicate/argument pair is associated with the minimal subtree that includes the word sequences of them both, hereafter called PAF. For example, in Figure 5, the substructures inside the three frames are the semantic/syntactic structures associated with the three arguments of the verb *to bring*, i.e. S_{Arg0}, S_{Arg1} and S_{ArgM}.

It is worth to note that PAF aims to capture all the information between a predicate and one of its arguments. PAF is quite intuitive and, to conceive it, the designer requires much less linguistic knowledge about semantic roles than those necessary to manually define effective features. The main drawback of its use is that important structural information, i.e. inter-argument dependencies, is neglected.

The large PropBank corpus makes the learning via tree kernels quite time consuming. Consequently, the algorithms presented in the previous section are very useful to speed up the learning/classification processes and make the kernel based approaches more applicable. The next section empirically shows the benefit of our DAG-based algorithms.

VI. Experiments

These experiments aim to show that our approach based on DAGs provides two kinds of benefits to the perceptron algorithms: a much faster computational time and a much lesser memory requirement.

For such purpose, we measured the computation time and the memory allocation for both the traditional Perceptron algorithm and the one based on DAGs. The target learning tasks were those involved in SRL. This is usually divided in two classification steps: argument boundary detection and argument classification. In the former step, all the nodes of the sentence parse tree are classified in *correct* or *incorrect* boundaries. The *correct* label means that the leaves (i.e. words) of the tree rooted in the target node are all and only those constituting an argument. In the latter step, given a

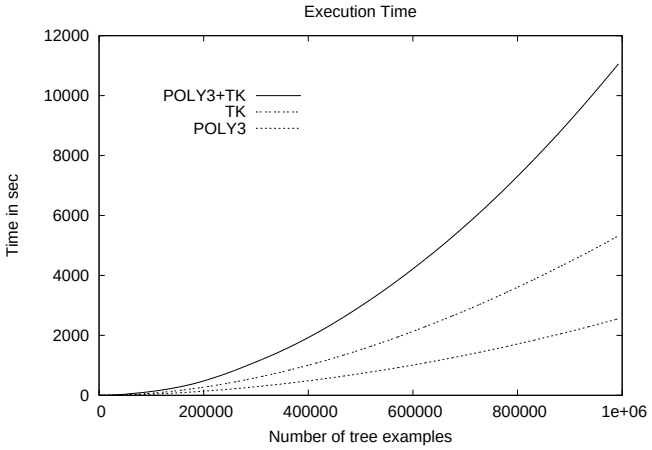

Fig. 6. Execution time for various kernels.

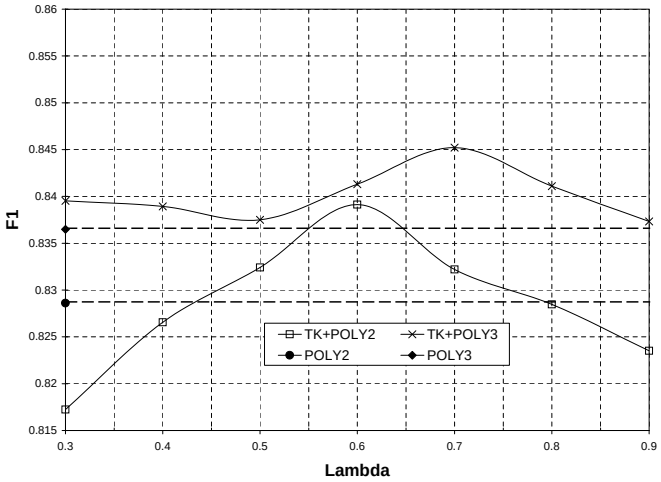

Fig. 7. The F1 of POLY2 and POLY3 and plots of the F1 of their combination with TK.

correct boundary node (i.e. an argument node), its type, i.e. Arg0, Arg1,..,Arg5, ArgA and ArgM, is determined.

As a referring dataset, we used PropBank (`www.cis.upenn.edu/~ace`) along with PennTree bank 2 [15]. This large corpus contains about 53,700 sentences annotated with predicative information.

In our experiments, we concentrated on boundary detection as the number of classifying instances is much larger. Indeed, they include all parse-tree nodes. For these experiments, we used the first 7 sections of PennTree bank for training and Section 24 for testing (in line with many systems of CoNLL SRL shared task 2005) for a total of 71,523 positive and 921,296 negative examples in training and 7,705 positive and 108,104 negative examples in testing.

As the DAG performance is affected by node distribution within trees along with their maximum and average outdegree, we have studied such characteristics in our data sets. Table I reports statistics about the data derived from the boundary detection dataset. We note that there is a large number of relatively small trees which however can have a large out-degree. Globally, the amount of nodes that have to be processed is very large, thus, the dataset is suitable to demonstrate the computational efficiency of our approach.

	Training	Test
Number of trees	992,819	115,809
Total number of nodes	14,365,253	1,686,167
Average number nodes in a tree	14.47	14.56
Average maximum outdegree	2.32	2.33
Max outdegree	15	15

TABLE I
FEATURES OF SYNTACTIC TREES IN THE BOUNDARY DETECTION DATASET.

The experiments were carried out with our implementation of the models proposed in sections III and IV. These were applied to the PAF trees as well as the standard feature vectors described in Section V.

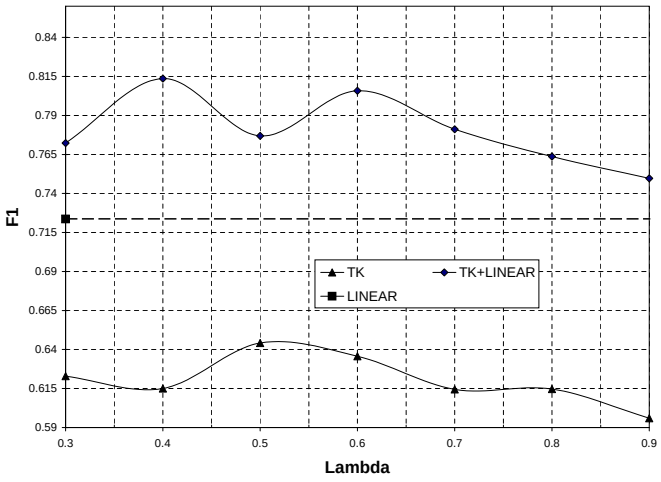

Fig. 8. The F1 of LINEAR and plots of the F1 of TK and TK+LINEAR.

A. Results

In these tests the DAG approach was applied to a very complex task from an efficiency point of view. Indeed, the number of instances of the boundary dataset was about one million. Such dataset has been prohibitive for computational expensive approaches like Support Vector Machines (see [4]): only using a polynomial kernel on standard features (in general much faster than tree kernels), they required 10 days to converge. On the contrary, as it can be seen in figure 6 for three different choice of kernels, the execution time never exceeds 4 hours on a common PC (Intel Core 2 Duo E6400 2.13GHz), and allowed us to experiment with (a) different kernel combinations, i.e. a tree kernel on PAF trees and polynomial kernels on *standard features* and (b) different values of λ hyperparameter.

Concerning storage requirements, no more than 450MB of memory has been allocated by any of our experiment.

Figures 7 and 8 report the plots of the F1-measure of

Polynomial kernel of degree q (POLYq), tree kernel (TK) and their combination (TK+POLYq) according to different λ values. We note that:

- POLY, in line with [16] reaches the highest $F1$, with degree 3. Also there is a large gap between linear and a degree larger than 1. This because the combination of standard features is quite important [16].
- TK shows a much lower F1 than POLYq (for any degree). This is due to some limits of the used PAF, i.e. as the boundary of an argument can be associated with several PAFs, the generalization task of a learning machine[2] increases.
- TK+POLY3 achieves the very high $F1$ of 84.5 ($\lambda = .7$) which is close to the $F1$ reached by the most accurate (among 20 participants) systems in the SRL shared task of CoNLL 2005. For example, by applying SVMs on the same dataset (on standard features), we obtained (after 10 days of processing) an F1 of 86.7.

In summary, given the high efficiency of the DAG algorithm, we could apply TK+POLY model to a very large dataset and correctly parameterize our system whereas the time required by SVMs (with just a polynomial kernel) is unaffordable. Moreover, the gap in accuracy between SVMs and DAGs can be reduced if we re-apply the latter to the training data. For example, using $\lambda = 0.7$ and iterating two times the DAG-Perceptron algorithm, we obtained an F1 of 85.7, i.e. only 1 percent point less.

VII. Conclusions

Kernel methods are effective approaches to the modeling of structured objects in learning algorithms. The major drawback is their typically high computational complexity.

To alleviate such problem, we have proposed Direct Acyclic Graphs to reduce the computational burden and storage requirements by representing common structures and feature vectors. This paper shows that substantial computational savings can be obtained for the perceptron algorithm using tree and polynomial kernels over a quite extensive dataset made available by the PropBank project.

The experiments on one million of instances show that our model makes use of kernels for trees practical for real applications. From the accuracy point of view, the possibility of using large amount of data allowed us to reach the state-of-the-art on automatic detection of Semantic Role Labeling as defined in the CoNLL shared task.

It is important to stress that the basic idea of DAGs 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 [6], boosting [21] and Bayes point machines [9].

[2]Accuracy similar to the one reached by linear kernel can be obtained by marking the argument node in the structure.

Moreover, it is possible to improve the basic perceptron algorithm when multiple trees, fed from several input stream of examples, are available simultaneously. For this situation, a simple variant of the basic perceptron which updates the model in parallel can be applied. Finally, there are ways to further exploit the shared sub-structures when gathering testing trees.

References

[1] G. AdelsonVelskii and Y. Landis, "An information organization algorithm," *Translation in NASA document n63-11777*, 1963.

[2] F. Aiolli, G. Da San Martino, A. Moschitti, and A. Sperduti, "Fast on-line kernel learning for trees," in *ICDM*, Hong Kong, China, 2006.

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

[4] X. Carreras and L. Màrquez, "Introduction to the CoNLL-2005 shared task: Semantic role labeling," in *Proceedings of CoNLL-2005*, 2005.

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

[6] C. Cortes and V. Vapnik, "Support-vector networks." *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[7] D. Gildea and D. Jurasfky, "Automatic labeling of semantic roles," *Computational Linguistic*, vol. 28, no. 3, pp. 496–530, 2002.

[8] D. Haussler, "Convolution kernels on discrete structures," University of California, Santa Cruz, Tech. Rep. UCSC-CRL-99-10, July 1999.

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

[10] T. Jaakkola, M. Diekhans, and D. Haussler, "A discriminative framework for detecting remote protein homologies," *Journal of Computational Biology*, vol. 7, no. 1,2, pp. 95–114, 2000.

[11] P. Kingsbury and M. Palmer, "From Treebank to PropBank," in *Proceedings of LREC'02*, Las Palmas, Spain, 2002.

[12] J. Kivinen, A. J. Smola, and R. C. Williamson, "Online learning with kernels," *Signal Processing, IEEE Transactions on*, vol. 52, no. 8, pp. 2165–2176, 2004.

[13] 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)*, 2004, pp. 152–160.

[14] C. Leslie, E. Eskin, A. Cohen, J. Weston, and W. S. Noble, "Mismatch string kernels for discriminative protein classification," *Bioinformatics*, vol. 20, no. 4, pp. 467–76, 2004.

[15] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The Penn Treebank," *Computational Linguistics*, vol. 19, pp. 313–330, 1993.

[16] A. Moschitti, "A study on convolution kernels for shallow semantic parsing," in *Proceedings of ACL'04*, Barcelona, Spain, 2004.

[17] ——, "Making tree kernels practical for natural language learning," in *Proceedings of EACL'06*, Trento, Italy, 2006.

[18] S. Pradhan, K. Hacioglu, V. Krugler, W. Ward, J. H. Martin, and D. Jurafsky, "Support vector learning for semantic argument classification," *Machine Learning Journal*, 2005.

[19] F. Rosemblatt, "A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[20] H. Saigo, J. Vert, T. Akutsu, and N. Ueda, "Protein homology detection using string alignment kernels," *Bioinformatics*, vol. 20, pp. 1682–1689, 2004.

[21] R. E. Schapire and Y. Singer, "Improved boosting algorithms using confidence-rated predictions." *Machine Learning*, vol. 37, no. 3, pp. 297–336, 1999.

[22] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

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

[24] C. Watkins, "Dynamic alignment kernels," Royal Holloway, University of London, Tech. Rep. CSD-TR-98-11, January 1999.