Noname manuscript No.
(will be inserted by the editor)

Fast Support Vector Machines for Convolution Tree Kernels

Aliaksei Severyn · Alessandro Moschitti

Received: date / Accepted: date

Abstract Feature engineering is one of the most complex aspects of the system design in machine learning. Kernel methods provide the designer with formidable tools to tackle such complexity. Among others, tree kernels (TKs) have been successfully applied for representing structured data in diverse domains, ranging from bioinformatics and data mining to natural language processing. One drawback of such methods is that the learning typically requires a large number of kernel computations (quadratic in the number of training examples) between the model and training examples. However, in practice substructures often repeat in the data which makes it possible to avoid a large number of redundant kernel evaluations.

In this paper, we propose the use of Directed Acyclic Graphs (DAGs) to compactly represent trees in the training algorithm of Support Vector Machines (SVMs). In particular, we use DAGs for each iteration of the cutting plane algorithm (CPA) to encode the model composed by a set of trees. This enables DAG kernels to efficiently evaluate TKs between the current model and a given training tree. Consequently, the amount of total computation is reduced by avoiding redundant evaluations over shared substructures. We provide theory and algorithms to formally characterize the above idea, which we tested on several datasets. The empirical results confirm the benefits of the approach in terms of significant speedups over previous state-of-the-art methods. In addition, we propose an alternative sampling strategy within the CPA to address the class-imbalance problem, which coupled with fast learning methods provides a viable TK learning framework for a large class of real-world applications.

Aliaksei Severyn · Alessandro Moschitti
Department of Computer Science and Engineering
University of Trento
Via Sommarive 5, 38123 POVO (TN) - Italy
E-mail: {severyn, moschitti}@disi.unitn.it

1 Introduction

Structural kernels have been successfully applied to ease the design of machine learning systems in diverse domains, ranging from bioinformatics [19, 21,30] and datamining [3,38,33,45,42,7,48,44] to Natural Language Processing (NLP) [20,9,34,18,10,4]. They free oneself from tedious manual feature engineering by automatically generating huge feature spaces assuming that the learning algorithm will end up using the most relevant features for a given task. This is especially useful when designing models for domains where no expert knowledge is easily available with respect to which features are most useful for a given problem. However, the use of kernel methods has been restricted to relatively small datasets as the training becomes much slower compared to linear models. Indeed, the major drawback of kernel methods, Support Vector Machines (SVMs) in particular, is the necessity to carry out learning in the dual space, where training complexity is typically quadratic in the number of training instances. This is largely attributed to the fact that the model weight vector is represented as a linear combination of training examples (support vectors) that all lie in the implicit feature space spanned by a given kernel function. As the size of the training set increases the number of support vectors in the kernel expansion of the model also tends to grow linearly [37]. Thus, evaluating a dot product between a model and a given example entails a large number of kernel computations over the training examples in the model.

Recently, a number of efficient methods to train SVMs based on the idea of the Cutting Plane Algorithm (CPA) have been proposed [16,12]. The CPA finds the model parameter vector by iteratively constructing cutting plane models that refine the estimation of the empirical risk. The optimal solution is a linear combination of such cutting planes. The linear-time behavior of the CPA again depends on the possibility to compact the model by summing up its constituent feature vectors such that the dot product can be computed efficiently. Unfortunately, again, for the reason briefly outlined above the method scales well only when linear kernels are used. To address slow learning with non-linear kernels, Joachims and Yu [17] propose to extract basis vectors to compactly represent cutting plane models, which speeds up both classification and learning. However, this requires to solve a non-trivial optimization problem, which renders intractable when considering discrete feature spaces generated by structural kernels. Finding a set of basis vectors in such high-dimensional spaces produced by arbitrary kernels, and in particular structural kernels, is an active research area.

Another approach of adapting CPA for non-linear kernels by reducing the number of kernel evaluations is studied in [46], where sampling is used to reduce the number of basis functions in the resulting kernel expansion. In [31], we showed that the same algorithm can be successfully applied to SVM learning with structural kernels on very large data obtaining speedup factors up to 10 over conventional SVMs. The approach was rather general as we did not make any assumption on the data. In contrast, in [1,2], we exploited a specific approach based on Directed Acyclic Graphs (DAGs) in the online

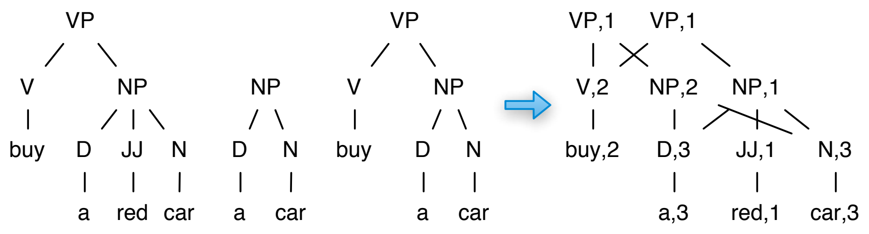


Fig. 1 Three syntactic trees and the resulting DAG.

learning setting to speed up the perceptron algorithm. DAGs were used to compactly represent tree forests given by the support vectors found by the learning algorithm. The approach reduced the number of expensive kernel evaluations, since DAGs provided the means to avoid redundant computations over shared substructures.

To give an intuition of the DAG approach for compact model representation studied in this article, consider an example of the tree-structured data, e.g. syntactic parse trees that are used extensively in Natural Language Processing. Fig. 1 shows a model consisting of only three syntactic trees[1] on the left and the resulting DAG on the right. As we can see, the subtree of the noun phrase `[NP [D a][N car]]` is repeated in two trees, thus the frequency of the corresponding node is updated to 2. Also smallest subtrees such as `[D a]` and `[D car]` are shared with a frequency of 3. The two subtrees rooted in `VP` are different and require different roots but they can still share some of their subparts, e.g. `[V buy]`.

In [32], we modeled DAGs to encode the cutting plane models computed at each iteration of the CPA algorithm. We presented two different algorithms, which, by compressing the trees in the CPA model, delivered impressive speedups for both training and testing. However, consecutive experiments revealed that as the size of the training data becomes larger, the speedup with respect to the algorithm using vanilla CPA model decreases, requiring a more thorough study.

In this paper, we extended and assessed our previously proposed methods by also deriving insights that better explain our approach. In particular, our contributions are as follows: firstly, we have extended our approach to any tree kernel satisfying some properties, e.g., our approach can be applied to a more general tree kernel, namely, the Partial Tree Kernel (PTK) [23]. PTK not only enables the use of dependency syntactic trees and other different syntactic paradigms, but also allows for applying our fast approach to many other application domains, e.g., it can be applied to XML trees or any other tree-structured data. In contrast to Syntactic Tree Kernel (STK) used in the previous study, PTK takes on a more fine-grained approach by matching any subsequence of children nodes of a given node. This necessitates modifications

[1] node labels define syntactic categories: NP - noun phrase, D - determiner, V - verb, VP - verb phrase, N - noun, JJ - adjective

in the organization of the DAG structure and also yields different levels of compression compared to the STK. We extensively study the effects of using PTK when compacting tree forests into DAGs.

Secondly, we investigated the speedup decrease by defining a new efficiency measure based on atomic kernel operations that in our case is the Δ function (i.e., the evaluation of the number of shared substructures rooted at two given nodes). This allowed us to exactly verify the speedup independently of the hardware used for running the experiments.

Next, we included a parallelization approach and a method for handling class-imbalanced datasets in the CPA algorithm, which has been previously missing. Plain CPA model previously studied in [31] simply learns a model that minimizes the error rate, which in the case of imbalanced datasets tends to produce models with low Recall. While, CPA can be used to train models that optimize various performance measures, e.g. F_1 score [15], this, however, entails the use of non-decomposable loss-functions, which, in turn, requires to compute the inner product over the entire training set when constructing cutting plane models at each iteration of the CPA. Hence, it prevents the use of sampling to speed up the learning with non-linear kernels. Conversely, we show that a simple cost-proportionate sampling technique is an elegant solution to extend the CPA to handle class-imbalanced data. We demonstrate that using an alternative sampling strategy within the CPA to build cutting planes at each iteration, indeed, provides an efficient way to tune up Precision and Recall of the obtained classifer. We also show that the original convergence bounds still apply to the modified algorithm.

Finally, we carried out an extensive evaluation of our approaches on five datasets: (a) a large dataset of Semantic Role Labeling (SRL) that contains a collection of parse trees expressing predicate argument relationships; (b) a new dataset derived from the previous one by removing lexical information, i.e. words, (unlexicalized trees); (c) question classification dataset, i.e., a taxonomy of the question types used in question answering systems; (d) question and answer pairs from Yahoo! answers; and (e) a new dataset from INEX [39] 2005 competition that contains a collection of XML trees. We evaluated the speedup in terms of the training time and the number of Δ-iterations for both STK and the newly proposed PTK on DAGs.

The results show that (i) our approach defined in [32] generalizes to most of tree-based kernels; and (ii) the high speedup achieved in [32] was also due to the compactness of the model, which could better fit in the CPU cache, amplifying the benefit of our approach. Nevertheless, the results also demonstrate that there is still a significant speedup for any size of the data at hand. Additionally, when dealing with less sparse data, e.g., with unlexicalized trees, the impact of our approach is further amplified. In particular, on the INEX data, whose trees show much larger repetition of the sub-structures, the DAG methods deliver the speedups of about two orders of magnitude w.r.t. to plain CPA model. This reveals that the potential impact of our approach may be beyond those we have outlined here.

In the remainder of this article sections 2 and 3 provide the reader with the required background on the CPA algorithm and structural kernels. Section 4 illustrates how the idea of compacting trees into a DAG can be used to speed up the CPA algorithm. Section 5 demonstrates our approach to construct DAGs from a tree forest, the computation of the DAG tree kernel and the parallelization of the resulting CPA algorithm. Section 6 illustrates our methods for dealing with class imbalance. Section 7 illustrates an extensive empirical study on five different datasets. Finally, section 9 derives the conclusions outlining future research directions.

2 Preliminaries: Cutting Plane Algorithm with Sampling

In this section, we begin by briefly introducing the problem formulation for binary SVMs to ease the illustration of a re-elaborated version of the cutting plane method (originally proposed in the context of structural SVMs) for binary classification. Having considered the case for linear SVMs, we point out the main source of inefficiency for the case when non-linear kernels are used. Next, we present the idea of using sampling [46] to approximate cutting planes computed at each iteration of the CPA, which is shown to alleviate high training costs for SVMs with non-linear kernels.

2.1 Cutting-plane algorithm (primal)

Given a dataset of n labeled examples $X = \{(x_i, y_i)\}_{i=1}^n$ with $x_i \in \mathcal{X}$ and $y_i \times \{-1, +1\}$, binary SVMs seek to find a linear decision function $f(\mathbf{w}, \mathbf{x}) = sign(\mathbf{w} \cdot \mathbf{x})^2$ that given a test example $\mathbf{x}$ and the model weight vector $\mathbf{w} \in \mathcal{X}$ predicts its label. The model weight vector $\mathbf{w}$ is estimated by solving the following optimization problem:

$$\begin{aligned} \underset{\mathbf{w},\boldsymbol{\xi}\geq 0}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + \frac{C}{n}\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}\cdot\mathbf{x}_i) \geq 1 - \xi_i,\ i = 1, \ldots, n \end{aligned} \tag{1}$$

where the first term in the objective function is a regularizer encoding the maximum-margin principle and the second term represents the empirical loss incurred on the training set. The constraints enforce the requirement to classify the training examples with a minimum margin. The slack variables ξ_i allow for violations in classification, which is useful in case of the noisy data. The margin trade-off parameter C controls the balance between the regularization term and the empirical loss.

[2] here we fix the bias term b at zero, as it could be easily incorporated in feature vectors as an additional constant

Algorithm 1 Cutting Plane Algorithm (primal)

```
1: Input: $X = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, $C$, $\epsilon$
2: $S \leftarrow \emptyset; t \leftarrow 0$
3: repeat
4:     $(\mathbf{w}, \xi) \leftarrow$ optimize OP2 over the constraints in $S$
5:     for $i = 1$ to $n$ do
6:         $c_i^{(t)} \leftarrow \begin{cases} 1 \textbf{ if } y_i(\mathbf{w} \cdot \mathbf{x}_i) \leq 1 \\ 0 \textbf{ otherwise} \end{cases}$
7:     $d^{(t)} \leftarrow \frac{1}{n} \sum_{i=1}^n c_i^{(t)}$
8:     $\mathbf{g}^{(t)} \leftarrow \frac{1}{n} \sum_{i=1}^n c_i^{(t)} y_i \mathbf{x}_i$
9:     $S \leftarrow S \cup \{(d^{(t)}, \mathbf{g}^{(t)})\}$
10:    $t \leftarrow t + 1$
11: until no CPMs are violated by more than $\epsilon$
12: return $\mathbf{w}, \xi$
```

Next, we consider an equivalent formulation of the SVM training problem, known as a 1-slack reformulation [16], to derive a more efficient version of the CPA for binary classification:

$$\begin{aligned} &\underset{\mathbf{w},\xi\geq 0}{\text{minimize}} && \frac{1}{2}\|\mathbf{w}\|^2 + C\xi \\ &\text{subject to} && \frac{1}{n}\sum_{i=1}^{n} c_i y_i \mathbf{w}\cdot\mathbf{x}_i \geq \frac{1}{n}\sum_{i=1}^{n} c_i - \xi,\ \forall \mathbf{c} \in \{0,1\}^n \end{aligned} \tag{2}$$

where a binary vector $\mathbf{c} = (c_1, \ldots, c_n) \in \{0,1\}^n$ is an index into the training set and selects which training examples form a given constraint. Hence, each of such constraints is composed by a linear combination of the constraints of the form: $y_i(\mathbf{w} \cdot \mathbf{x}_i) \geq 1 - \xi_i$. The key idea behind this equivalent reformulation is to represent the empirical risk by only a single slack variable ξ shared across all the constraints. Even though the number of slack variables is reduced to only a single ξ, the number of constraints is 2^n (as defined by all possible values of $\mathbf{c}$). This prevents the application of off-the-shelf optimization methods to directly solve optimization problem in Eq. 2 (OP2). Nevertheless, it has been shown in [40] that the cutting plane algorithm applied to the OP2 uses only a small subset of active constraints that is independent of the size of the training set.

To solve OP2, an adaptation of the generic cutting plane algorithm [16] for binary classification problem has been shown to yield significant performance gains over conventional classifiers. The CPA is presented in Alg. 1. It starts with an empty set of constraints S and computes the optimal solution to the OP2. Next, the algorithm forms a binary vector $\mathbf{c}$ that is merely an index into the training set and selects which training examples will form the next cutting plane model (CPM) (defined by an offset $d^{(t)} = \frac{1}{n}\sum_{i=1}^n c_i^{(t)}$ and a gradient $\mathbf{g}^{(t)} = \frac{1}{n}\sum_{i=1}^n c_i^{(t)} y_i \mathbf{x}_i$ (lines 5-8)). The cutting plane model encodes a constraint $\mathbf{w} \cdot \mathbf{g}^{(t)} \geq d^{(t)} - \xi$ that is violated the most by the current solution $\mathbf{w}$, which is then included in the set of active constraints S (line 9). This process is repeated until no CPMs are violated by more than ϵ (line 11), which is formalized by the following criteria $\mathbf{w} \cdot \mathbf{g}^{(t)} \geq d^{(t)} - \xi + \epsilon$.

2.2 Cutting-plane algorithm (dual)

While SVMs discussed in the previous section seek to build classifiers that are linear functions, one may achieve better accuracy by using the power of kernels to build highly discriminative non-linear decision boundaries. This is achieved by introducing a mapping function $\phi(\cdot)$ that projects the inputs into some high-dimensional feature space. However, the use of kernels requires to solve the OP2 in the dual space. Its solution $\mathbf{w}$ lies in the feature space defined by a kernel $K(\mathbf{x}_i, \mathbf{x}_k) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_i)$. Omitting the details, it can be verified (by deriving the dual from OP2) that the solutions of the primal and dual problems are connected via:

$$\mathbf{w} = \sum_{j=1}^{t} \alpha_j \mathbf{g}^{(j)}, \tag{3}$$

where α_i are dual variables, $\mathbf{g}^{(j)} = \frac{1}{n} \sum_{k=1}^{n} c_k^{(j)} y_k \phi(\mathbf{x}_k)$ denotes the gradient of the cutting plane model added at iteration j and t is the current iteration.

As one can see, with the use of kernels the gradient $\mathbf{g}^{(j)}$ represents a weighted sum of training examples that lie in the feature space spanned by $\phi(\cdot)$. This implies that a dot product between $\mathbf{w}$ and a given example $\mathbf{x}_i$ requires an explicit computation with each of its components encoded by $\mathbf{g}^{(j)}$, i.e., a common trick, to compact $\mathbf{w}$ into a single vector by simply summing up its n feature vectors is no longer possible. This prohibits to exploit linear-time training algorithms of SVMs with linear kernels and represents a major bottleneck of kernelized SVMs. We will address the problem of compact representation of the cutting plane models in Section 4.

Computing an inner product between the weight vector $\mathbf{w}$ and an example $\mathbf{x}_i$ involves the sum of kernel evaluations for each example $\mathbf{x}_k$ in the cutting plane model $\mathbf{g}^{(j)}$ over the set S. In particular, using the expansion of $\mathbf{w}$ from (3), the inner product required to find the next cutting plane model (steps 5-8 in the Alg. 1), renders as:

$$\mathbf{w} \cdot \phi(\mathbf{x}_i) = \sum_{j=1}^{t} \alpha_j \mathbf{g}^{(j)} \cdot \phi(\mathbf{x}_i) = \sum_{j=1}^{t} \alpha_j \Big(\frac{1}{n} \sum_{k=1}^{n} c_k^{(j)} y_k \Big) K(\mathbf{x}_k, \mathbf{x}_i), \tag{4}$$

The analysis of the inner product given by (4) reveals that the number of kernel evaluations is $O(tn)$. Indeed, the number of non-zero elements in each $\mathbf{g}^{(j)}$ is proportional to the number of support vectors which grows linearly with the training size n [37]. Performing the kernel evaluations for each cutting plane model $\mathbf{g}^{(j)}$ in the set S, we obtain the complexity of (4) is $O(tn)$. Since the inner product (4) needs to be computed for each training example (lines 5-6 in Alg. 1) we obtain the total $O(tn^2)$ scaling behavior for each iteration of the Alg. 1.

The obtained quadratic scaling in the number of examples makes cutting plane training for non-linear SVMs prohibitively expensive for even medium-sized datasets and no better than convenvional decomposition methods such as SMO or SVM-light. To address this limitation [46] proposed to construct approximate cuts by sampling r examples from the training set. The idea is to replace the expensive computation of the cutting plane (lines 5-7, Alg. 1) over all training examples n by a sum over a smaller sample r, s.t. the number of examples in $\mathbf{g}^{(j)}$ is reduced from $O(n)$ to $O(r)$. In this case the double sum of kernel evaluations in (4) reduces from $\sum_{i,j=1}^{n} K(\mathbf{x}_i, \mathbf{x_j})$ to a more tractable in practice $\sum_{i,j=1}^{r} K(\mathbf{x}_i, \mathbf{x_j})$. This reduces the complexity of each iteration of Alg. 1 from $O(tn^2)$ to $O(tr^2)$.

While using sampling to approximate cutting planes computed at each iteration introduces an additional parameter into the learning algorithm, it has been shown in [46] that the resulting training and test set errors are stable with respect to changes in the sample size r. Additionally, [31] extensively studied the effects of the sample size on the obtained runtime speedups within the context of SVMs with structural kernels. It has been shown that selecting smaller sample sizes r (as small as 100 examples) provides significant savings in the runtime while leading to only a small loss in accuracy.

3 Learning from Structured Data with Tree Kernels

In this section we introduce tree kernels that represent trees in terms of their sub-structures (fragments). The kernel function detects if a tree subpart (common to both trees) belongs to the feature space that we intend to generate. For such purpose, the desired fragments need to be described and efficiently computed. We consider two important convolution kernel functions: the syntactic tree kernel (STK) and the partial tree kernel (PTK).

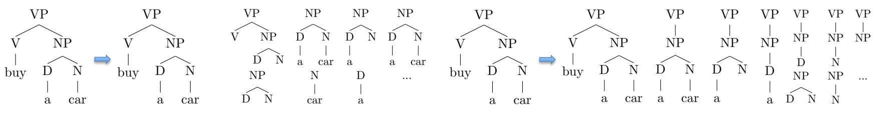
3.1 Counting shared fragments in convolution kernels

Convolution TKs compute the number of common substructures between two trees T_1 and T_2 without explicitly considering the whole fragment space. For this purpose, let the set $\mathcal{T} = \{t_1, t_2, \ldots, t_{|\mathcal{T}|}\}$ be the space of substructures and $\chi_i(n)$ be an indicator function, equal to 1 if the target t_i is rooted at a node n and equal to 0 otherwise. A tree-kernel function over T_1 and T_2 is

$$TK(T_1, T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2), \tag{5}$$

where N_{T_1} and N_{T_2} are the sets of the T_1's and T_2's nodes, respectively and

$$\Delta(n_1, n_2) = \sum_{i=1}^{|\mathcal{T}|} \chi_i(n_1) \chi_i(n_2). \tag{6}$$



(a) Syntactic Tree fragments (STFs) (b) Partial Tree fragments (PTFs)

Fig. 2 Examples of different classes of tree fragments used as features by Tree Kernels.

which computes the number of common fragments rooted in the n_1 and n_2 nodes.

Of course, the number above depends on how fragments are defined. Indeed, there are different types as described in [23]. We consider three important characterizations: (i) the *subtrees*, (ii) the syntactic tree fragments (STFs) and (iii) the partial tree fragments (PTFs). These three types of tree fragments determine three different kernel functions.

A *subtree* rooted in a node n of a target tree T is a substructure that includes n with all of its descendants. A generalization of subtrees are STFs that do not necessarily include all the descendants of n, although each of its nodes contain exactly the same edges of T. For example, Figure 2(a) shows 10 STFs (out of total 17) of the subtree rooted in `VP` (of the left tree). In phrase structure syntactic trees, the constraint on the edges, is equivalent to impose that grammatical rules cannot be broken[3]. For example, `[VP [V NP]]` is an STF, which has two non-terminal symbols, `V` and `NP`, as leaves whereas `[VP [V]]` is not an STF, i.e. the rule `VP->V NP` can not be split.

If we relax such constraint, we obtain more general substructures called PTFs. These can be generated by the application of partial production rules of the grammar, consequently `[VP [V]]` and `[VP [NP]]` are valid PTFs. More in general, nodes in PTFs can have any subset of edges that had in T. This means that PTFs are not constrained to any grammar and can be applied to any tree structure of any domain. Figure 2(b) shows that the number of PTFs derived from the same tree as before is higher (i.e. 30 PTs).

3.2 Syntactic Tree Kernel (STK)

The Δ function depends on the type of fragments that we consider as *basic* features. To evaluate the number of STFs, we can use the following algorithm:

1. if the productions at n_1 and n_2 are different then $\Delta(n_1, n_2) = 0$;
2. if the productions at n_1 and n_2 are the same, and n_1 and n_2 have only leaf children (i.e. they are pre-terminal symbols) then $\Delta(n_1, n_2) = 1$;
3. if the productions at n_1 and n_2 are the same, and n_1 and n_2 are not pre-terminals then

$$\Delta(n_1, n_2) = \prod_{j=1}^{nc(n_1)} (1 + \Delta(c_{n_1}^j, c_{n_2}^j)) \tag{7}$$

[3] Any tree can be seen as generated by an underlying grammar where the production rules are given by a node (left handside) and its children (right handside).

where $nc(n_1)$ is the number of children of n_1 and c_n^j is the j-th child of the node n. Note that, since the productions are the same, $nc(n_1) = nc(n_2)$.

$\Delta(n_1, n_2)$ evaluates the number of STFs common to n_1 and n_2 as proved in [8]. Moreover, a decay factor λ can be added by modifying steps (2) and (3) as follows[4]:

2. $\Delta(n_1, n_2) = \lambda$,
3. $\Delta(n_1, n_2) = \lambda \prod_{j=1}^{nc(n_1)} (1 + \Delta(c_{n_1}^j, c_{n_2}^j))$.

The computational complexity of STK is $O(|N_{T_1}| \times |N_{T_2}|)$ but as shown in [24], the average running time tends to be linear, i.e. $O(|N_{T_1}| + |N_{T_2}|)$, for natural language syntactic trees.

It should be noted that STK was devised for processing syntactic trees. Its main characteristic is that the production rules of the grammar used to generate the tree will not be *broken* to generate fragments. This corresponds to the restriction of not separating children in the related substructures (i.e. STFs). STK can be applied to compute the similarity measure between any trees (not necessarily syntactic parse trees) with a restriction that any node of the generated STF must still contain all (or none) of its children.

3.3 Partial Tree Kernel (PTK)

The computation of PTFs is carried out by the Δ function defined as follows:

1. if the node labels of n_1 and n_2 are different then $\Delta(n_1, n_2) = 0$;
2. else:

$$\Delta(n_1, n_2) = 1 + \sum_{\mathbf{I}_1, \mathbf{I}_2, l(\mathbf{I}_1) = l(\mathbf{I}_2)} \prod_{j=1}^{l(\mathbf{I}_1)} \Delta(c_{n_1}(\mathbf{I}_{1j}), c_{n_2}(\mathbf{I}_{2j}))$$

where $\mathbf{I}_1 = \langle h_1, h_2, h_3, .. \rangle$ and $\mathbf{I}_2 = \langle k_1, k_2, k_3, .. \rangle$ are index sequences associated with the ordered child sequences c_{n_1} of n_1 and c_{n_2} of n_2, respectively. $\mathbf{I}_{1j}$ and $\mathbf{I}_{2j}$ point to the j-th child in the corresponding sequence, and, again, $l(\cdot)$ returns the sequence length, i.e. the number of children.

Furthermore, we add two decay factors: μ for the depth of the tree and λ for the length of the child subsequences with respect to the original sequence, which accounts for gaps. Hence, the expression for the Δ function for PTK derives as follows:

$$\Delta(n_1, n_2) = \mu\Big(\lambda^2 + \sum_{\mathbf{I}_1, \mathbf{I}_2, l(\mathbf{I}_1) = l(\mathbf{I}_2)} \lambda^{d(\mathbf{I}_1) + d(\mathbf{I}_2)} \prod_{j=1}^{l(\mathbf{I}_1)} \Delta(c_{n_1}(\mathbf{I}_{1j}), c_{n_2}(\mathbf{I}_{2j}))\Big), \quad (8)$$

where $d(\mathbf{I}_1) = \mathbf{I}_{1l(\mathbf{I}_1)} - \mathbf{I}_{11} + 1$ and $d(\mathbf{I}_2) = \mathbf{I}_{2l(\mathbf{I}_2)} - \mathbf{I}_{21} + 1$. This way, we penalize both larger trees and child subsequences with gaps. Eq. 8 is more

[4] To have a similarity score between 0 and 1, we also apply the normalization in the kernel space: $TK_{norm}(T_1, T_2) = \frac{TK(T_1, T_2)}{\sqrt{TK(T_1, T_1) \times TK(T_2, T_2)}}$.

general than Eq. 7. Indeed, if we only consider the contribution of shared subsequences containing all children of nodes, we actually obtain the STK kernel. The computational complexity of PTK is $O(p\rho^2|N_{T_1}| \times |N_{T_2}|)$ [23], where p is the largest subsequent of children that we want to consider and ρ is the maximal out-degree observed in the two trees. However, as shown in [23], the average running time again tends to be linear in the number of nodes for natural language syntactic trees.

4 Fast CPA for Structural Kernels

In this section we present an approach to significantly speed up the approximate CPA for structural kernels described in Section 2. We observe that for convolution structural kernels that are defined in terms of its substructures, the cutting plane model can be compactly represented as a Directed Acyclic Graph (DAG), where each unique substructure is stored only once. This helps to speed up both the training and classification as the repeating kernel evaluations over shared substructures can be avoided. Most interestingly, this approach can be parallelized during training, thus, making structural kernel learning practical on larger datasets.

4.1 Compacting cutting plane models using DAGs

In the previous section we have seen that computing a cutting plane model (CPM) at each iteration involves a quadratic number of kernel evaluations. Using sampling to approximate the cutting plane helps to reduce the number of kernel evaluations.

Here, we explore another method to reduce the number of kernel computations when convolution structural kernels are used. Indeed, when applied to structural data such as sequences, trees or graphs, we can take advantage of the fact that many examples share common sub-structures. Hence, we can use a compact representation of a cutting plane model to avoid redundant computations over repeating sub-structures. In particular, when dealing with tree-structured data, a collection of trees can be compactly represented as a DAG [2]. In the following we briefly introduce the idea behind using DAGs to compactly represent a tree forest and then show how it applies to speed up the learning algorithm.

4.2 DAG tree kernels

A DAG can efficiently represent a set of trees (a forest F) by including only unique subtrees and accounting for the frequency of the repeated substructures. Given the DAG representation previously presented in Fig. 1, we can define tree kernel functions between a DAG and a tree, which compute exactly the same kernel, with a relevant speedup. The support for this algorithm is given by the following:

Theorem 1 *Let* $\mathbf{D}$ *be a DAG representing a tree forest* F *and* $K_{dag}(\mathbf{D}, T_2) = \sum_{n_1 \in N_{\mathbf{D}}} \sum_{n_2 \in N_{T_2}} f(n_1)\Delta(n_1, n_2)$ *then*

$$\sum_{T1 \in F} TK(T_1, T_2) = K_{dag}(\mathbf{D}, T_2), \tag{9}$$

where $f(n_1)$ *is the frequency associated with* n_1 *in* $\mathbf{D}$*,* TK *is any tree kernel function that can be factorized with Eq. 5 and a* $\Delta(n_1, n_2)$ *function, which counts the number of shared subtrees rooted in* n_1 *and* n_2*.*

Proof Let $\mathcal{S}(F)$ be the set of possible *subtrees* (see the definition in Sec. 3.1) of F, i.e., the substructures whose leaves coincide with those of the original tree (in general $\mathcal{T} \neq \mathcal{S}$), then $\sum_{T_1 \in F} TK(T_1, T_2) = \sum_{T_1 \in F} \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2)$ $= \sum_{T_1 \in F} \sum_{\substack{n_1 : t \in \mathcal{S}(T_1) \\ n_1 = r(t)}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2)$, where $r(t)$ is the root of the subtree t. The last expression is equal to $\sum_{\substack{n_1 : t \in \mathcal{S}(F) \\ n_1 = r(t)}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2)$. Let $\mathcal{S}'$ be the unique subtrees of $\mathcal{S}$, we can rewrite the above equation as:
$\sum_{\substack{n_1 : t \in \mathcal{S}'(F) \\ n_1 = r(t)}} f(n_1) \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2) = \sum_{\substack{n_1 : t \in \mathbf{D} \\ n_1 = r(t)}} \sum_{n_2 \in N_{T_2}} f(n_1)\Delta(n_1, n_2) =$ $\sum_{n_1 \in N_{\mathbf{D}}} \sum_{n_2 \in N_{T_2}} f(n_1)\Delta(n_1, n_2)$ $\square$

Remark 1 It should be noted that no assumption is made on $\Delta(n_1, n_2)$, thus our approach is valid for a vast set of tree kernels defined by a specific form of its Δ function. Since STK and PTK are based on Eq. 5 and their $\Delta(n_1, n_2)$ function computes the number of substructures rooted in n_1 and n_2 the following holds:

Corollary 1 *Given a DAG* $\mathbf{D}$ *and a forest* F*, let us define*
1. $STK_{dag}(\mathbf{D}, T_2) = \sum_{n_1 \in N_{\mathbf{D}}} \sum_{n_2 \in N_{T_2}} f(n_1)\Delta_{STK}(n_1, n_2)$ *and*
2. $PTK_{dag}(\mathbf{D}, T_2) = \sum_{n_1 \in N_{\mathbf{D}}} \sum_{n_2 \in N_{T_2}} f(n_1)\Delta_{PTK}(n_1, n_2)$

then such DAG kernels exactly compute $\sum_{T1 \in F} STK(T_1, T_2)$ *and* $\sum_{T1 \in F} PTK(T_1, T_2)$*, respectively, where* Δ_{STK} *is Eq. 7 and* Δ_{PTK} *is Eq. 8.*

Remark 2 It is easy to prove that convolution kernels [13] can be factorized with Eq. 5 and a generic $\Delta(n_1, n_2)$. Therefore, our approach at least applies to such large class of kernels.

4.3 Fast Computation of the CPM on Structural Data

Having introduced the DAG tree kernel, we redefine the most computationally expensive part of the CPA, i.e. the inner product in Eq. 4 required to compute the CPM by compacting $\mathbf{g}^{(j)}$ into a single DAG model $\mathbf{D}^{(j)}$:

$$\mathbf{w} \cdot \phi(\mathbf{x}_i) = \sum_{j=1}^{t} \alpha_j \Big(\frac{1}{n} \sum_{k=1}^{n} c_k^{(j)} y_k\Big) K(\mathbf{x}_k, \mathbf{x}_i) = \frac{1}{n} \sum_{j=1}^{t} \alpha_j K_{dag}(\mathbf{D}^{(j)}, \mathbf{x}_i) \tag{10}$$

Unlike Eq. 4, where each cutting plane $\mathbf{g}^{(j)}$ is an arithmetic sum of training examples, here we take advantage of the fact that a collection of trees can be efficiently put into an equivalent DAG $\mathbf{D}^{(j)}$. As shown in Th. 1 computing a kernel $K_{dag}(\cdot,\cdot)$ between an example and a DAG that represents a collection of trees yields an exact kernel value. The benefit of such representation comes from the efficiency gains obtained by speeding up kernel evaluations over the sum of examples compacted into a single DAG.

Alternatively, to benefit even more from the compact representation offered by DAGs, we can put all the cutting planes from the active set S into a single DAG model $\widehat{\mathbf{D}}$, such that the inner product in Eq. 10 is reduced to a single kernel evaluation:

$$\mathbf{w} \cdot \phi(\mathbf{x}_i) = \frac{1}{n} \sum_{j=1}^{t} \alpha_j K_{dag}(\mathbf{D}^{(j)}, \mathbf{x}_i) = \frac{1}{n} K_{dag}(\widehat{\mathbf{D}}^{(t)}, \mathbf{x}_i), \tag{11}$$

where $\widehat{\mathbf{D}}^{(t)}$ at iteration t is built by compacting all $\mathbf{D}^{(j)}$ together with their corresponding dual variable α_j. This ensures that a single K_{dag} evaluation over the full DAG model makes Eq. 11 equivalent to computing a weighted sum of K_{dag} using individual $\mathbf{D}^{(j)}$ in Eq. 10. Henceforth, we call the approach to compute the inner-product in Eq. 4 using more efficient expressions in Eq. 10 and Eq. 11 as SDAG and SDAG+ respectively. A more detailed explanation of the DAG structure implemetation along with the pseudocode of algorithms for building a DAG from a set of trees, compacting a set of DAGs into a single DAG and computing K_{dag} are given in the forthcoming section.

Now we are ready to present the new cutting plane algorithm (Alg. 2) adapted for the use of tree kernels with DAGs. Different from Alg. 1 here we use a set of r examples uniformly sampled from the original training set to approximate the CPM computed at each iteration of the CPA, s.t. computing the inner product over all n examples in the training set is reduced to a much smaller sample r. This also reduces the size of the resulting model weight vector, since each CPM includes maximum only r training points. To compute the CPM we can use either SDAG or SDAG+ approach to form a binary vector $\mathbf{c}$, which then defines the training examples that are further inserted into the DAG to represent the CPM at iteration t.

As one can see, while using SDAG+ approach provides better compression, since all CPM models $\mathbf{D}^{(j)}$ are compacted into a single $\widehat{\mathbf{D}}^{(t)}$, it, however, needs to be re-built at each iteration to accommodate an update in vector $\boldsymbol{\alpha}$ after re-solving the dual of OP2. Nevertheless, the time to construct $\widehat{\mathbf{D}}$ is linear in the number of nodes in the model and imposes negligible computational overhead in practice. Another computational drawback of using full DAG model compared to the set of $\mathbf{D}^{(j)}$ is that in the former case we need to compute the update of the Gram matrix column (line 4 in Alg.2) $G_{it} = \mathbf{D}^{(i)} \cdot \mathbf{D}^{(t)}$ for $1 \leq i \leq t$, while in the latter case it is obtained automatically from computing Eq. 10.

Even though the worst-case complexity of computing CPMs at each iteration using both variants of DAGs is still $O(r^2)$, in practice we can observe much

Algorithm 2 Cutting Plane Algorithm (dual) using DAGs

```
1: Input: $X = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, $r$, $C$, $\epsilon$,
2: $S \leftarrow \emptyset$; $\mathbf{D} \leftarrow \emptyset$; $t \leftarrow 0$;
3: **repeat**
4:    **Update the Gram matrix $G$ with a new CPM**
5:    $\boldsymbol{\alpha} \leftarrow$ optimize Wolfe dual of OP2
6:    $I \leftarrow$ index set of $r$ examples uniformly sampled from the training set X
7:    $\mathbf{c}^{(t)} \leftarrow$ find CPM using SDAG (Alg. 3) or SDAG+ (Alg. 4)
8:    $\mathbf{D}^{(t)} \leftarrow$ buildDAG($\mathbf{x}_{c^{(t)}}, \mathbf{y}_{c^{(t)}}$)
9:    $d^{(t)} \leftarrow \frac{1}{r}\sum_{i=1}^{r} c_i^{(t)}$
10:   $S \leftarrow S \cup \{(d^{(t)}, \mathbf{D}^{(t)})\}$
11:   $t \leftarrow t + 1$
12: **until** no CPMs are violated by more than $\epsilon$
13: **return** $\mathbf{w}, \xi$
```

Algorithm 3 Find CPM with SDAG

```
Input: $\mathbf{x}, S, \boldsymbol{\alpha}, I$
**for each** $i \in I$ **do**
    $c_i^{(t)} \leftarrow \begin{cases} 1 & \textbf{if } y_i/r\sum_{j=1}^{t}\alpha_j K_{dag}(\mathbf{D}^{(j)}, \mathbf{x}_i) \le 1 \\ 0 & \textbf{otherwise} \end{cases}$
**return** $\mathbf{c}^{(t)}$
```

Algorithm 4 Find CPM with SDAG+

```
Input: $\mathbf{x}, S, \boldsymbol{\alpha}, I$
$\widehat{dag} \leftarrow$ compactDAG($S, \boldsymbol{\alpha}$)
**for each** $i \in I$ **do**
    $c_i^{(t)} \leftarrow \begin{cases} 1 & \textbf{if } y_i/rK_{dag}(\widehat{dag}, \mathbf{x}_i) \le 1 \\ 0 & \textbf{otherwise} \end{cases}$
**return** $\mathbf{c}^{(t)}$
```

better scaling behavior, since in real datasets examples tend to share many common substructures. As verified by the extensive experiments in section 7, this greatly speeds up both training and classification by avoiding redundant kernel computations. Finally, it is important to note that the obtained Alg. 2 preserves all theoretical benefits of the approximate CPA with sampling, since the kernel computations remain the same, while in practice greatly reducing the number of expensive kernel evaluations to compute the CPM.

5 Implementation of the DAG Kernel

The implementation of the DAG kernel requires two different algorithms for: (i) efficiently inserting trees into a DAG and (ii) computing a kernel between a DAG and a given tree. Additionally, the DAG kernel computation can be parallelized.

5.1 DAG construction

There are various methods to efficiently build a DAG corresponding to a collection of trees, F, see for example [2]. Given a tree, $T \in F$, we need to insert its nodes in the DAG, where the uniqueness of each node is defined by its corresponding subtree. The latter criteria is crucial, since the DAG representation has to conform to the recursive nature of the tree kernel functions computed over the DAG, i.e. two nodes which corresponding subtrees differ in only one element will have to be inserted separately in the DAG. For this purpose, to insert tree nodes $n \in T$, we need to be able to efficiently check if the subtree rooted at n is already in the DAG. The key of the node n can simply by a serialized string representation of the subtree rooted at this node. In case the node is already present in the DAG we only need to update its associated weight.

The node weights are defined as $y_j/\sqrt{|T|}$ or $y_j \cdot \alpha_j/\sqrt{|T|}$ for SDAG and SDAG+ respectively[5], such that the dag kernel produces an equivalent tree kernel value when computing the inner product in Eq. 10 or Eq. 11. Hence, each element in the DAG is a pair of two items: a node (we only need to keep a pointer) and its weight. This is enough to obtain a compact model representation, where repeating tree sub-structures are accounted by their corresponding weights and can be uniquely stored in the DAG. This construction ensures that one obtains the same TK values.

As we have seen from section 4.2 a DAG tree kernel is defined as the sum of Δ-function evaluations over the node pairs sharing some common property, i.e. production rules (STK) or node labels (PTK) are the same. For this purpose, it is convenient to maintain a second associative array indexed by either the production rule (STK) or node labels (PTK) in the DAG structure, such that for a given node $n \in T$ we can retrieve a list of all matching nodes N in the DAG in constant time.

Hence, we design the DAG data structure D to contain two associative arrays: *nodes* and *productions*. The former is used to perform constant time membership checks when inserting a new node, such that two nodes with identical subtrees are stored only once. The latter allows for the constant time retrieval of a list of node pairs in a DAG matching the production rule (node label) of a given node n, which is used for computing a DAG tree kernel. In this way we can efficiently insert trees into a DAG and enumerate all the candidate substructures sharing the same production (STK) or node label (PTK) to compute the tree kernel between a DAG and a given tree. The aforementioned implementation ideas are formalized in Algorithms 5 and 8 which provide the pseudocode for inserting trees into a DAG.

More specifically, to insert a tree T into a DAG D we proceed as follows: we first define the weight of the nodes to be inserted into a DAG as the tree label y_i divided by the tree norm $|T|$, such that we effectively compute the

[5] here, we consider the normalized version of the tree kernel, i.e. $K(T1, T2) = K(T1, T2)/(\sqrt{K(T1, T1)} * \sqrt{(K(T1, T1))})$, hence, we need to introduce the tree norm $|T|$ into the node weight.

inner product as defined in Eq. 10. Next, for each node in the input tree, we compute the nodes id[6] and use it to check if the node is already present in the DAG. If yes, we simply update its weight, otherwise we insert the node together with its weight into both associative arrays: *nodes* and *productions*. For the latter, we use nodes production (label) to retrieve a list of matching nodes and append the new element. Finally, Alg. 8 shows how a set of DAGs $\mathbf{D}^{(j)}$ are compacted into a single $\widehat{D}$, which allows for a reduction of an inner product in Eq. 10 to a single kernel evaluation as shown in Eq. 11. Note that, to account for individual α_j of each $\mathbf{D}^{(j)}$ when inserting its node into a $\widehat{\mathbf{D}}$, we simply make it as an additional factor of the node weight, i.e. $weight \cdot \alpha_j$ (line 7).

5.2 DAG kernel computation

Having discussed the particular implementation of the DAG data structure that allows for efficient DAG construction from a collection of trees, we now consider how one can compute a tree kernel over a given tree and the constructed DAG D. The associative array *productions* allows us to efficiently compute TKs by retrieving (in constant time) a list of nodes in the DAG matching a given node $n \in T$ to evaluate Δ for each pair of nodes. In particular, computing a tree kernel between a DAG and a tree (see Alg. 11) simply requires (i) looping over nodes in a tree (line 3), (ii) retrieving a list of nodes in the DAG matching node n (line 7) and (iii) summing up the product between the weight of the considered node in the DAG and the value returned by a call to a Δ function (line 10). Note that we return the final value of the *sum* divided by a tree norm $|T|$, s.t. evaluating K_{dag} over a DAG and a set of trees yields a normalized TK value. This procedure is similar for both STK and PTK kernels with the difference that for PTK we form the matching node pairs using node labels instead of production rules.

5.3 Parallelization

The modular nature of the CPA suggests easy parallelization. In fact, in our experiments, we observed that at each iteration 95% of the total learning time is spent on computing the CPM (steps 3-9, Alg. 2). This involves computing Eq. 10 over the set of individual DAGs or Eq. 11 using full DAG model for the sample of r training examples. Using p processors the complexity of this pre-dominant part can be brought down from $O(r^2)$ to $O(r^2/p)$.

[6] Computing node ids requires to serialize subtrees rooted at each node which is linear in the tree size and is done at the preprocessing stage.

Algorithm 6 Insert a tree node into a DAG

```
1: Input: sample of $I$, dag $D$
2: for $i \in I$ do
3:     $weight \leftarrow \frac{y_i}{\sqrt{|T_i|}}$
4:     for each $node \in T_i$ do
5:         addNode$(D, weight, node)$

   procedure addNode$(D, weight, node)$
   $key \leftarrow id(node)$
   $K \leftarrow$ all keys in $D.nodes$
   if $key \in K$ then
       updateWeight$(D.nodes[key], weight)$
   else
       $newDagElement \leftarrow$ construct a new pair $(weight, node)$
       $D.nodes[key] \leftarrow newDagElement$
       $dagElementsList \leftarrow D.productions[node.production]$
       append$(dagElementsList,\ newDagElement)$
```

Algorithm 7 Compact a set of CPMs into a single DAG

```
1: Input: set of CPMs $S$, vector of dual variables $\boldsymbol{\alpha}$
2: $\widehat{D} \leftarrow newDag()$
3: for $j = 1$ to $length(S)$ do
4:     $K \leftarrow$ all keys in $D^{(j)}.nodes$
5:     for each $key \in K$ do
6:         $weight, node \leftarrow D^{(j)}.nodes[key]$
7:         addNode$(\widehat{D}, weight \cdot \alpha_j, node)$
8: return $\widehat{D}$
```

Algorithm 8 Compute $K_{dag}(\mathbf{D}, \mathbf{T})$

```
1: Input: tree $T$, $dag$
2: $sum \leftarrow 0$
3: for each $node \in T$ do
4:     $key \leftarrow$ production rule of $node$
5:     $P \leftarrow$ all production rules in $D.productions$
6:     if $key \in P$ then
7:         $dagElementsList \leftarrow D.productions[key]$
8:         for each $dagElement \in dagElementsList$ do
9:             $weight, dagNode \leftarrow dagElement$
10:            $sum \leftarrow sum + weight \cdot \Delta(node, dagNode)$
11: return $sum/\sqrt{|T|}$
```

6 Handling Class-Imbalanced data

Having considered a set of techniques to speed up the training of SVMs with tree kernels, we now turn to addressing another important problem of dealing with class-imbalanced data. This problem often arises in situations when we have to deal with datasets where the number of negative examples largely outnumbers the number of positive examples. On such datasets, a typical classifier that is minimizing a mis-classification rate is likely to learn a model that will tend to label all examples as negative. Hence, it will will do poorly in terms of Precision and Recall.

Thus, in this section, we extend the theory of the cutting-plane algorithm to tackle class-imbalance problem. Our approach is based on an alternative sampling strategy, e.g. cost-proportionate sampling, that is effective for tuning up Precision and Recall on class-imbalanced data. Typically, cost-proportionate sampling is used to alter the distribution of the original training set to make the proportion of positive and negative examples balanced, such that the classifier can be trained on a balanced data. However, the CPA operates differently as it iteratively draws samples to build an approximation of the cutting plane models at each step. Hence, it is important to verify that altering the distribution of the examples in the sample used to build CPM at each step, indeed, allows for an effective way to tune up Precision and Recall of the final classifier. We also demonstrate that the same convergence bounds hold when cost-proportionate sampling is applied within the CPA.

6.1 Cost-proportionate sampling

Conventional SVM problem formulation allows for natural incorporation of example dependent importance weights into the optimization problem. We can modify the objective function to include example dependent cost factors:

$$\begin{aligned} \underset{w,\xi_i \geq 0}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + \frac{C}{n}\sum_i^n z_i\xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}\cdot\mathbf{x_i}) \geq 1 - \xi_i,\ 1 \leq i \leq n \end{aligned} \tag{12}$$

where z_i is the importance weight of example i and $\frac{1}{n}\sum_i^n z_i\xi_i$ serves as an upper bound on the total cost-sensitive empirical risk. This problem formulation where there is an individual slack variable ξ_i for each example is typically referred to as "n-slack" formulation.

In the dual space, the example-dependent costs captured by cost factors z_i translate into the box constraints imposed on each dual variables: $0 \leq \alpha_i \leq z_iC,\ 1 \leq i \leq n$ such that the z_iC sets an upper bound on the values of α_i. This feature to integrate importance weights z_i in the problem formulation is implemented in SVM-light software.

This natural modification of the quadratic problem, is, however, difficult to incorporate in the case of 1-slack formulation (OP2). Indeed, in the case of 1-slack formulation we have a single slack variable ξ that is shared among all the constraints. More importantly, moving to the dual space, the box constraints $0 \leq \alpha_i \leq C$ are no longer for each individual dual variable but for a sum: $\sum_i \alpha_i$. This makes the 1-slack problem formulation difficult to incorporate importance weights directly. Nevertheless, the idea of approximating the cutting plane model at each iteration via sampling suggests a straightforward solution.

Indeed, we can extend the original CPA to the case of cost-sensitive classification. A straight-forward way to do this is instead of using uniform sampling to build an approximation to the CPM at each iteration in Alg. 2), we can draw examples according to their importance weights using the cost-proportionate rejection sampling technique (Alg. 9).

Algorithm 9 Cost-proportionate rejection sampling

```
1: Pick example $(\mathbf{x_i}, y_i, z_i)$ at random
2: Flip a coin with bias $z_i/Z$
3: if heads then
4:     keep the example
5: else
6:     discard it
```

Here z_i is the importance weight of the i-th example and Z is an upper bound on any importance value in the dataset. This process is repeated until we sample the required number of examples r. This modification enables the control over the proportion of examples from different classes that will form a sample used to compute the CPM.

Unlike the conventional approaches for addressing the class-imbalance problem, that either under-sample the majority class or over-sample the minority class from the training data, the rejection sampling coupled with CPA does not completely discard examples from the training set. At each iteration it forms a sample according to the pre-assigned importance weights for each example, such that examples from both the majority and minority classes enter the sample in the desired proportion. This process is repeated until the algorithm converges. Thus, the learner has the chance to incorporate relevant information present in the data over a number of iterations before it converges. This way, the method preserves the global view on the dataset and no relevant information is lost during the iterative optimization process unlike in the "one-shot" sampling methods.

Another benefit of this approach is that by increasing the importance weight of the minority class, we give its examples more chance to end up in CPM and hence, become support vectors. This way the imbalanced support-vector ratio is automatically tuned to include more examples from the minority class, which gives more control over the class-imbalance problem. Proving this property could be an interesting theoretical result.

6.2 Theoretical Analysis of the Algorithm

Cost proportionate rejection sampling allows for natural extension of the binary classification to importance weighted binary classification. It achieves this task by re-weighting the original distribution of examples D according to the importance weights of examples such that the training is effectively carried out under the new distribution $\hat{D}$.

In [47] it is shown that by transforming the original distribution D to a training set under $\hat{D}$, one can effectively train a **cost-insensitive** classifier on a dataset $\hat{D}$ such that it will minimize the expected risk under the original distribution D.

Theorem 2 (Translation Theorem; [47]) *Learning a classifier h to minimize the expected cost-sensitive risk under the original distribution D is equivalent to learning a decision function to minimize the expected* ***cost-insensitive*** *risk under the distribution $\hat{D}(x,y,z) \equiv \frac{z}{E_{(x,y,z)\sim D}[z]} D(x,y,z)$.*

The proof is a straight-forward application of the definitions and simply follows by establishing an equivalence relationship between the expected cost-sensitive risk $E_{(x,y,z)\sim D}[z\Delta(y,h(x))]$ under the original distribution D and the expected cost-insensitive risk $E_{(x,y,z)\sim \hat{D}}[\Delta(y,h(x))]$ under the transformed distribution $\hat{D}$. The theorem produces an important implication that by transforming the original distribution D to $\hat{D}$ according to example-dependent importance weights, a classifier for the cost-sensitive problem over D can be obtained with a cost-insensitive learning algorithm over D. We can use this finding to show that the convergence proof for the original CPA with uniform sampling naturally applies to the proposed version of the algorithm that uses cost-proportionate rejection sampling:

Theorem 3 (Convergence) *Assume $R = max_{1\le i\le n}\|\phi(\mathbf{x}_i)\|$, i.e. R is an upper bound on the norm of any $\phi(\mathbf{x}_i)$, and $\Delta = max_{1\le i\le n}\|\ \Delta(y,y_i)\|$, the number of steps required by Alg. 2 using the sampling strategy of Alg. 9 is upper bounded by $8C\Delta R^2/\epsilon^2$.*

Proof. We first note that the cost-proportionate rejection sampling (Alg. 9), used to build the approximate cutting plane model, at each step re-weights the original distribution D according to the importance weights of the examples. This means that we are effectively training a cost insensitive classifier that draws examples to build the cutting plane model from the transformed distribution $\hat{D}$. By invoking the Translation Theorem (2), we establish that, to obtain a cost-sensitive classifier that minimizes the expected risk under the original distribution D, it is sufficient to learn a cost-insensitive classifier under the transformed distribution $\hat{D}$. The CPA that draws examples from D using rejection sampling is equivalent to the original CPA applying uniform sampling to the transformed distribution $\hat{D}$. Thus, we can reutilize the proof in [46] of the convergence bounds for the original CPA with uniform sampling over $\hat{D}$. This states that CPA with uniform sampling terminates after at most $8C\Delta R^2/\epsilon^2$ iterations. By applying such bound, we have proved the thesis of the theorem.
Remarks. The main idea to obtain convergence bounds in [46] is to set an upper bound on the value of the dual objective and if there exists a lower bound on the minimal improvement of the dual objective at each iteration, then the algorithm will terminate in a finite number of steps.

Indeed, using the relationship between primal and dual problems, we have that a feasible solution of the primal OP1, such as, for example: $\mathbf{w} = \mathbf{0}$, $\xi = \Delta$, forms an upper bound $C\Delta$ on the dual objective of 2. Next, in [40] it is shown that the inclusion of ϵ-violated constraint at each iteration improves the dual objective by at least $\epsilon/8R^2$. Since the dual objective is upper bounded by $C\varDelta$, the algorithm terminates after at most $8C\varDelta R^2/\epsilon^2$ iterations.

The derivation of the bound on the minimal improvement of the dual objective obtained at each step only depends on the values of ϵ and R and does not rely on the assumption about distribution of the examples. Also note that each cutting plane model built via rejection sampling is a valid constraint for the OP1.

7 Experiments

In our experiments we pursue a three-fold goal: (i) study the effects of compacting the cutting plane model by using DAGs on both training and classification runtime; (ii) demonstrate the speedup factors one can obtain after straightforward parallelization offered by the CPA; and (iii) demonstrate the ability of the cost-proportionate sampling scheme to tune up Precision and Recall;

7.1 Experimental setup

We integrated CPA with uniform sampling as described in [46] within the framework of SVM-Light-TK [24,14] to enable the use of structural kernels, e.g. we used STK and PTK (see Sec. 3). PTK has been indicated as the most accurate in similar tasks, e.g. [24], while PTK is a more general yet much more computationally demanding. To measure the classification performance, we use Precision, Recall and F_1-score, i.e. a harmonic mean between Precision and Recall.

For the DAG implementation, we employ highly efficient Judy arrays[7]. For brevity, we refer to the CPA with uniform sampling as uSVM; uSVM where each cutting plane $\mathbf{g}^{(j)}$ is compacted into a $\mathbf{D}^{(j)}$ as SDAG; uSVM with a single DAG that fits all active constraints in the set S as SDAG+; uSVM with cost-proportionate sampling as uSVM+j (Alg. 9), and SVM-light-TK as SVM. The margin trade off parameter is fixed at 1.0.

We ran all the experiments on machines equipped with Intel® Xeon® 2.33GHz CPUs carrying 6Gb of RAM under Linux. Parallel implementation relies on the OpenMP library.

7.2 Data and models

To evaluate the efficiency of the compact model representation offered by SDAG and SDAG+ algorithms with respect to uSVM, we use Semantic Role Labeling (SRL) benchmark. The dataset consists of the Penn Treebank texts [22], PropBank annotation [28] and Charniak parse trees [6] as provided by the

[7] http://judy.sourceforge.net

CoNLL 2005 shared task on Semantic Role Labeling [5]. The goal is to recognize semantic roles of the target verbs in a given sentence.

SRL is a complex tasks where the state-of-the-art systems achieve F_1 at about 80%, which indicates the importance of extracting the best features. A common approach to tackle SRL problem involves two steps: (i) detection of the verb arguments and (ii) classification of identified arguments into their respective semantic categories and final annotation of the original parse tree. In our experiments, we focus on the first task of argument identification (i.e. the exact sequence of words spanning an argument). This corresponds to the classification of parse tree nodes into correct or not correct boundaries. For this purpose, we train a binary *Boundary Classifier* using the AST subtree defined in [25], i.e. the minimal subtree, extracted from the sentence parse tree, including the predicate and the target argument nodes. To evaluate the learned models we report the F_1[8] on two sections: 23 and 24, that contain 230k and 150k examples respectively. SRL dataset has already been used to extensively test uSVM for structural kernels and we follow the same setting as described in [31] unless mentioned otherwise.

We also conduct an important study on the sparsity effect inherent to the syntactic parse trees. While the trees from SRL dataset have a very small number of unique non-terminal nodes (less than 100), the number of unique subtrees is huge. This is largely attributed to a great variety of the leaf nodes (lexicals) which for syntactic parse trees are simply words. Consequently, many subtrees that have identical structure up to the leaf nodes may differ because only one word is different. Hence, such nodes when inserted into a DAG will be stored separately, which prevents greater levels of compression. In NLP, the number of unique leaf nodes (words) can be hundreds of thousands and more, therefore, to better understand the sparsity effect caused by the leaf nodes, we carry out another set of experiments on unlexicalized trees from SRL. We simply remove the leaf nodes and re-train our models on this modified data. This allows for better assessment of how our approach may perform in other settings where trees are much less diverse and DAGs can yield better compression.

Additionally, to better understand how the DAG idea translates to other domains different from NLP, we also present the results for XML tree classification from the INEX 2005 challenge [11]. The dataset is formed by XML documents describing movies from the IMDB site[9]. The particular steps taken to pre-process the trees are described in [39]. The size of the training and test set is 4820 and 4810. The distinctive property of this data is that the total number of unique node labels (XML tags) is 197. This is much smaller when compared to very sparse syntactic structures in NLP. This fact has a direct effect on the number of common substructures shared among the training ex-

[8] the reported scores corresponds to the accuracy of the binary classifier, which is slightly higher than the accuracy of the overall boundary detection due to errors in parsing.

[9] http://www.imdb.com

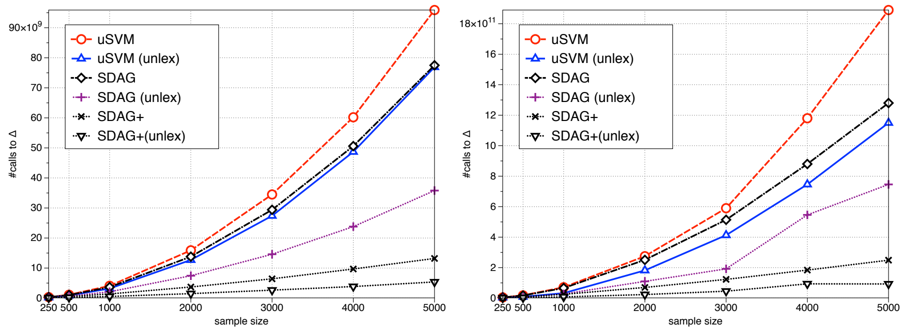


Fig. 3 Comparison of the number of calls to Δ for uSVM, SDAG, and SDAG+ using STK (left) and PTK (right) kernels on the original SRL and unlexicalized version.

amples. As we will see in the next section this plays a significant role for the SDAG and SDAG+ algorithms, which can exploit more compact models.

In the next set of experiments to study the ability of uSVM+j to tune up Precision and Recall we used two different natural language datasets: TREC 10 QA[10] (training: 5,483, test: 500) and Yahoo! Answers (YA)[11](train: up to 300k, test: 10k) to perform two similar tasks of QA classification. The task for the first dataset is to select the most appropriate type of the answer from a set of given possibilities. The goal of the experiments on these relatively small datasets is to demonstrate that rejection sampling is able to effectively handle class imbalance similar to SVM. For Yahoo! Answers dataset the classification task was set up as follows. Given pairs of questions and corresponding answers learn if in a given pair the answer is the "best" answer for a question. The goal of this experiment is to have a large classification task (300k examples in our experiments) to demonstrate that class-imbalance problem can be handled effectively at a scale where SVM becomes too slow.

7.3 Comparative speedup analysis of DAG-based model

The goal of this set of experiments is to study computational savings that come from using a compact representation of individual (SDAG) or the full set (SDAG+) of cutting plane models in S. As the baseline for the learning and classification runtime comparison, we use plain uSVM algorithm. Note that the classification accuracy is not of concern here (hence, not reported) since SDAG and SDAG+ produce exact kernel evaluations, thus they train the same model as uSVM.

Learning speedups. To carry out training, we use 100k examples from the second section of the SRL dataset. Figure 3 provides the first intuition on the

[10] http://l2r.cs.uiuc.edu/cogcomp/Data/QA/QC/
[11] retrieved through the Yahoo! Webscope program.

Table 1 Evaluation using STK: speedups w.r.t. training time and number of calls to Δ function for SDAG and SDAG+ over uSVM on three datasets: SRL (top), unlexicalized SRL (middle), and INEX 2005 dataset (bottom). The size of both SRL datasets is 100k whereas the size of INEX 2005 is 5k. For uSVM we report the absolute values of Δ and training time (minutes), t, while for SDAG and SDAG+ we illustrate the relative speedups, i.e., the rate between Δ and t and Δ_S and t_S of SDAG or Δ_{S+} and t_{S+} of SDAG+.

	uSVM		SDAG		SDAG+	
SAMPLE	Δ	t	Δ/Δ_S	t/t_S	Δ/Δ_{S+}	t/t_{S+}
			SRL			
250	3.3E+08	3	1.1	5.1	1.8	4.0
500	1.1E+09	10	1.1	6.3	2.2	5.4
1000	4.1E+09	37	1.1	7.0	3.0	7.8
2000	1.6E+10	138	1.1	7.3	4.3	11.0
3000	3.5E+10	303	1.2	7.2	5.4	14.3
4000	6.0E+10	517	1.2	7.2	6.2	17.2
5000	9.6E+10	834	1.2	7.6	7.3	20.0
			SRL (UNLEXICALIZED)			
250	2.3E+08	2	1.2	8.1	3.4	8.8
500	8.6E+08	6	1.4	8.3	4.4	11.5
1000	3.2E+09	21	1.5	8.6	5.9	16.0
2000	1.3E+10	85	1.7	8.8	8.6	19.8
3000	2.7E+10	183	1.9	8.8	10.3	22.3
4000	4.9E+10	325	2.0	9.1	12.7	26.7
5000	7.7E+10	513	2.1	9.2	14.3	28.9
			INEX 2005			
100	1.7E+08	1	13.1	7.5	21.7	6.9
250	8.6E+08	3	25.3	14.6	55.2	24.9
500	3.0E+09	8	43.1	23.1	90.4	43.3
1000	1.2E+10	31	78.8	46.0	156.4	77.5

runtime savings provided by SDAG and, especially, SDAG+ that is able to provide the most compact model representation. The graph plots the total number of calls to Δ function made by the learning algorithm during the training phase for uSVM, SDAG and SDAG+ for both the original SRL and SRL with lexicals removed. While the number of kernel evaluations technically remains the same for all the algorithms (using the same tree kernel), it is the number of Δ calls (see Eq. (7) and Eq. (8)) that greatly differ between uSVM, SDAG and SDAG+. As we can see, SDAG+ provides much better computational savings in terms of the Δ-calls than uSVM and SDAG for both STK and PTK. This benefit becomes especially strong for unlexicalized dataset where DAGs are able to provide even more compact representation.

Tables 1 and 2 present a more detailed performance comparison of SDAG and SDAG+ with respect to uSVM on 100k subset of SRL and its unlexicalized version where leaf nodes representing words (lexicals) were removed. The bottom part shows the results of training on INEX dataset. We carried out comparative experiments on both STK and PTK kernels and used uSVM outcome as a yardstick. For each algorithm and each kernel, we report two relative metrics: ratios between the number of calls to Δ function as defined in equations (7) and (8) for STK and PTK correspondingly.

Table 2 Evaluation using PTK kernel (see the caption of in Table 1).

	uSVM		SDAG		SDAG+	
SAMPLE	Δ	t	Δ/Δ_S	t/t_S	Δ/Δ_{S+}	t/t_{S+}
			SRL			
250	4.8E+09	23	1.0	1.5	1.7	1.2
500	1.8E+10	90	1.1	1.5	2.1	1.4
1000	7.1E+10	340	1.1	1.5	2.9	1.8
2000	2.7E+11	1781	1.1	2.1	3.9	3.3
3000	5.9E+11	3696	1.1	2.1	4.8	5.1
4000	1.2E+12	5039	1.3	2.1	6.4	5.7
5000	1.9E+12	6687	1.5	2.1	7.6	6.5
			SRL (UNLEXICALIZED)			
250	4.5E+09	20	1.8	1.7	2.9	2.3
500	8.8E+09	38	1.4	1.8	3.5	2.3
1000	3.3E+10	144	1.2	1.7	3.9	3.1
2000	1.8E+11	804	1.7	2.3	7.7	4.1
3000	4.1E+11	1405	2.1	2.2	9.1	5.4
4000	7.5E+11	2501	1.4	2.3	8.0	6.3
5000	1.0E+12	2822	1.5	2.4	10.8	7.2
			INEX 2005			
100	2.3E+09	1	90.7	12.8	144.8	13.4
250	1.3E+10	8	171.6	26.5	283.4	27.4
500	3.3E+10	21	145.3	21.9	243.9	22.3
1000	1.4E+11	82	152.4	21.4	260.4	21.4

As one can see both SDAG and SDAG+ deliver significant speedups during the learning with respect to uSVMs. The main quantity to observe here is the savings in Δ-computations as they largely define the runtime of the algorithms. SDAG+ performs much better than SDAG being able to provide the most compact representation of tree forests. The results become much stronger when considering unlexicalized SRL and especially INEX datasets where subtrees tend to be less sparse. SDAG+ is a clear winner here for both STK and PTK kernels. Another metric reported in Tables 1 and 2 is the actual training time. Surprisingly, SDAG+ is able to deliver speedups in training time up to 20 for STK kernel when a large sample size is used. We also observe that training time speedups are much higher than the savings in Δ-computations. This can be explained by the fact that the compact DAG model require less memory and can remain in the CPU cache thus delivering time savings better than expected[12]. Another interesting finding is that as subtrees become less sparse (unlexicalized SRL and INEX) we obtain much better compression. Especially, the results on INEX data where the number of unique node labels (XML tags) is much smaller than for natural language trees, show the true potential of compressing learning models into equivalent DAGs.

Classification experiments. Regarding classification, we compare SDAG+ with uSVM (see Table 3). We carry out learning for various sizes of the training set

[12] The exact quantification of the role of the CPU cache along with the design of more efficient algorithms based on it is beyond the scope of this paper.

Table 3 Classification speedups for SDAG+ over uSVM using STK kernel. Testing on 10k subset when learning on SRL subsets of varying size (1st column). Time indicated in seconds; comp denotes ratio between the number of nodes in SDAG+ and uSVM models; #SVs- number of support vectors.

	uSVM			SDAG+			
Data	#SVs	nodes	t	nodes	t_{S+}	comp	t/t_{S+}
10k	1686	33516	10.8	6350	0.5	5.3	24.0
25k	3392	67840	40.8	14212	1.2	4.8	33.2
50k	5876	117520	82.1	25506	2.9	4.6	28.3
75k	7489	149780	111.7	33552	5.5	4.5	20.5
100k	8674	215764	130.6	39787	6.7	5.4	19.5
250k	11234	224680	172.5	62094	16.8	3.6	10.2
500k	13037	260740	199.0	79978	26.6	3.3	7.5
750k	13270	265400	205.9	91048	33.9	2.9	6.1
1mil	13912	278240	216.3	97447	39.8	2.9	5.4

and perform testing on 10k of data. The values of interest here are the number of nodes in the final model and the testing time. As we can see, as the size of the training set increases not only the model becomes larger but also the nodes that end up in the model become sparser. This affects the compression rate of SDAG+ w.r.t. uSVM which results in smaller speedups for larger data.

Finally, in the Table 4, we report the results of comparison on 100k of SRL data between SVM, uSVM, SDAG, and SDAG+. This replicates the same setting as in [31]): the sample size is relatively small, i.e., 1k and 5k for 100k and 1 million datasets respectively. Therefore, the test conditions do not emphasize the benefits if the DAG models. Nevertheless, SDAG and SDAG+ algorithms deliver high speedups w.r.t. to uSVM and it becomes much larger when compared to SVM. Thus, DAG compression even on relatively sparse trees from SRL (compared to INEX dataset) as carried out in this experiment delivers very significant computational savings over conventional SVMs. We believe that applying it to very large XML document classification datasets would deliver even higher speedups against widely used in this setting SVM-light algorithm.

Table 4 Comparison of SVM, uSVM, SDAG and SDAG+ on 100k and 1mil SRL using STK kernel. For 100k the sample size was fixed at 1000 and for 1mil is 5000 (to replicate the experiment in [31]). The number of iterations is 300. The reported values are training time in minutes; values in parenthesis are relative speedups w.r.t. SVM.

size	SVM	uSVM	SDAG	SDAG+
100k	214	37 (6)	5 (41)	5 (45)
1mil	10705	814 (13)	349 (31)	264 (41)

Table 5 Handling class-imbalance problem on TREC 10 (top) YA (bottom). Ratio - proportion of negative examples w.r.t. positive; P/R - precision (P) and recall (R). The bottom row in YA is the performance using bag-of-words features on 75k subset. The reported F_1 scores that do not pass the statistical significance test (with $p \leq 0.05$) are marked by ^{1} for uSVM+j/uSVM and by ^{2} for uSVM+j/SVM.

TREC 10

DATA	RATIO	uSVM		uSVM+J		SVM	
		F-1	P/R	F-1	P/R	F-1	P/R
ABBR	1:60	87.5	100.0/77.8	84.2^2	80.0/88.9	84.2	80.0/88.9
DESC	1:4	96.1	95.0/97.1	96.1^{12}	95.0/97.1	94.8	97.7/92.0
ENTY	1:3	72.3	91.8/59.6	79.1	79.6/78.7	80.4	82.2/78.7
HUM	1:3	88.1	98.1/80.0	90.3	94.9/86.2	87.5	88.9/86.2
LOC	1:3	81.4	96.6/70.4	87.0	87.5/86.4	82.6	86.5/79.0
NUM	1:5	86.0	98.9/76.1	91.2	96.1/86.7	89.9	98.9/82.3

YAHOO ANSWERS

10K	1:1.5	37.4	33.5/42.2	39.1	29.6/57.7	37.9	24.2/87.7
50K	1:2.0	36.5	36.0/36.9	40.6	30.0/62.5	39.6	25.7/86.9
100K	1:2.4	33.4	36.2/31.1	40.2	30.2/59.9	40.3	26.6/83.5
150K	1:2.8	33.5	36.9/30.7	41.0	30.2/64.0	-	-
300K	1:3.4	23.8	40.1/16.9	41.4	30.7/63.8	-	-
BOW	1:2.0	34.2	33.2/35.3	38.1	27.5/61.7	36.3	22.5/93.5

7.4 Tuning up Precision and Recall.

To measure if the difference in the observed values of F_1 scores of the compared models is statistically significant we employed the implementation [27] of the assumption-free randomization framework [26]. The conclusion about the statistical significance of the difference in F_1 scores of considered models is made by assessing how likely the difference in the randomly shuffled predictions of two models is due to chance. We used the default number of 10,000 shuffles for each measurement.

We first report experimental results on question classification corpus on six different categories in Table 5 (since the dataset is small, we only report the accuracy). For both uSVM and uSVM+j, we fixed the sample size to 100. For uSVM+j, we picked the value of j from $\{1, 2, 3, 4, 5, 10\}$ and use the best results obtained on the validation set. For SVM, we carried out tuning of j parameter on a validation set. It is important to note that such parameter has slightly different meaning for uSVM+j and SVM. For the former, it controls the bias to reject negative examples during sampling (Alg. 9) to compute CPM, while for the latter it defines the factor by which training errors on positive examples outweigh errors on negative examples.

Analyzing the results from Table 5 (top), we can see that uSVM algorithm that uses uniform sampling obtains high Precision, as it minimizes the training error dominated by examples from negative class. This results in lower values of the Recall. Its rather high F_1 for ABBR dataset shows that the model simply misclassifies the examples from the minority class saturating the Precision. On the other hand, uSVM+j is able to establish a much better balance between Precision and Recall resulting in high F_1 scores across the majority of categories. Also the performance of SVM with the optimal set of parameters suggests that our method has a better capacity to control the imbalance problem than SVM. This can be explained by the fact, as suggested in [43], that $z_i C$ imposes only an upper bound on dual variables α_i, which results in poorer flexibility to control the class-imbalance with the j parameter of SVM.

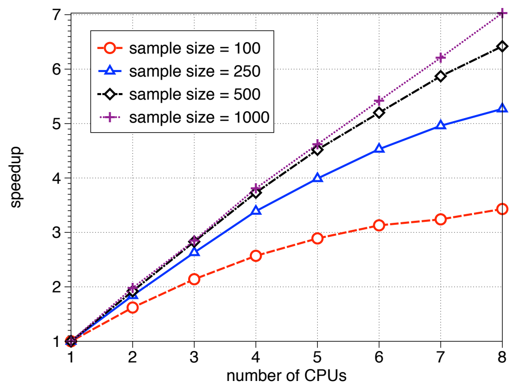
The results on Yahoo! Answers are displayed in Table 5 (bottom). For uSVM and uSVM+j, we fix the sample size at 500. Due to the constant time scaling behavior of uSVM [46], the training time for both uSVM and uSVM+j was slightly less than 10 hours across all subsets reported here. While being faster on small subsets of 5k, 10k and 25k, SVM begins to scale poorly on the subsets larger than 50k. Indeed, as studied in [46,31], CPA with sampling begins to outperform SVM starting from datasets of moderate size (around 50k in our experiments). SVM did not finish the training within 5 days for 150k and 300k subsets, hence there are missing values. We set the value of j parameter for uSVM+j equal to the ratio of negative to positive examples. This natural setting of j parameter for uSVM+j is driven by the intuition to make the distribution of examples from different classes approximately balanced inside each sample, such that the classifier learns on a balanced data. As one can see, this gives much better trade-off between Precision and Recall compared to uSVM. Looking at the results of SVM, we conjecture that here j parameter, similar to the results in previous experiments, is not flexible enough to deliver the optimal P/R trade-off. Also note that training SVM on 100k subset requires almost 4 days, which makes uSVM+j a viable tool for advanced text classification on large datasets, where obtaining optimal balance between Precision and Recall is hindered by the class imbalance problem.

The bottom row of Table 5 reports the results using bag-of-words (BOW) feature representation on 75k subset. We note that STK delivers an interesting 12% of relative improvement over BOW model on SVM. However, the main goal of this experiment was not to obtain the top classification performance on such noisy web data but rather to demonstrate that uSVM+j can efficiently deal with large imbalanced data.

7.5 Parallelization

To assess the effects of parallelization, we tested parallel versions of SDAG and SDAG+ on 50k subset of Yahoo! Answers dataset using up to 8 CPUs. The achieved speedups over the sequential algorithm are reported in Figure 4, where each curve corresponds to runtimes using different sample sizes: {100, 250, 500, 1000}. Increasing the sample size leads to the increase of the time spent to compute CPM, which makes the speedup achieved by parallelization for large sample sizes even more significant. Using the maximum number of 8 CPUs, we are able to achieve the speedup factor of about 7.0 (using sample size equal to 1000). Since classification can also be easily parallelized, we could experiment with larger sample sizes to obtain a more accurate model.

Fig. 4 Speedups due to parallelizing SDAG/SDAG+ on 50k Yahoo! Answers dataset.



8 Related work

To improve the scaling properties of SVM-light, a number of efficient algorithms using CPA-based algorithms have been proposed. For example, SVM^{perf} [16] exhibits linear computational complexity in the number of examples when linear kernels are used. While CPA-based approaches deliver state of the art performance w.r.t. accuracy and training time, they scale well only when linear kernels are used. The problem of efficient kernel learning for CPA has been studied in [17], where cutting plane models are compacted by extracting basis vectors. This, however, leads to a non-trivial optimization problem when arbitrary kernel functions are applied.

Regarding learning with structural kernels, compact representation of tree forests offered by DAGs was applied for speeding up training of the voted perceptron algorithm in [2]. Another interesting idea of hash kernels for structured data is proposed in [36], where hashing can generate explicit vector representation such that linear learning methods can be applied. However, it is likely that hashing all possible substructures generated by STK, which is exponential in the tree length, will make the preprocessing step too expensive. Also, due to hash collisions, this method computes approximate kernel values and its implications on the accuracy need to be studied more extensively.

A highly efficient subtree kernel on graphs that exploits the idea from Weisfeiler-Lehman test of graph isomorphism is proposed in [35]. While, it has been shown to work well on various graph datasets from bioinformatics, the subtree feature space generated by this kernel is inferior to more general STK and PTK, as its feature generation mechanism includes uniformly all the nodes in the neighborhood of a currently considered node within a given radius, i.e. it does not allow for incomplete tree fragments.

In [29], a more principled feature extraction algorithm to linearize TK spaces has been proposed. Its soundness is justified by the norm-preservation of the model learned by an SVM to extract the most relevant features. The post-analysis of the most relevant tree fragments extracted by the algorithm on the SRL task reveals that fairly complex structures with long term dependencies between the sentence constituents are pertinent for the SVM learner. This suggests that naïve feature enumeration coupled with feature hashing to reduce the effective dimension of the resulting feature vectors will result in lower performance on complex tasks such as SRL, were a few complex features provide essential discriminative power to the classifier.

Concerning class-imbalance problem for SVM learning, the most widely adopted method is to introduce different cost factors in the objective function s.t. the training errors for positive and negative examples receive different penalties [41]. This approach is implemented as the j option in SVM-light [14] that has a super-linear scaling behavior, which prohibits its use on large datasets. Our approach to accomplish cost-sensitive classification shares the idea of reductions put forward in [47] together with the benefit of the conventional approach in SVMs [41] to incorporate importance weights directly into the optimization process.

9 Conclusions and Future Work

In this paper we have presented several techniques to make learning with SVMs and convolution tree kernels applicable to a larger set of real-world applications. Firstly, we have defined a generalized theory and methods for using DAG kernels in the CPA algorithm with sampling. We have proved that our approach can be applied to any tree kernel computable by summing over $\Delta(n_1, n_2)$, where n_1 and n_2 are pairs of nodes from two trees (Th.1).

Secondly, we verified the theory above by modeling and implementing two algorithms: SDAG and SDAG+. The former compresses only the current CPM whereas the latter compacts the entire set of CPMs built so far during the learning of CPA. Both algorithms were used with STK and PTK that clearly satisfy the hypothesis of Th.1.

Thirdly, as PTK considers any node-child subsets to represent trees, we modified the organization of the DAG structure which results in different levels of compression. Consequently, we extensively studied the effects of using PTK when compacting tree forests into DAGs. In particular, we analyzed the efficiency of our algorithm based on the number of calls to Δ function to exactly verify the speedup independently of the hardware used in the experiments.

Additionally, we also included a parallelization approach and a method for handling imbalanced datasets in SDAG and SDAG+.

Finally, we have experimented with the models above on four datasets: (i) two versions of SRL data, lexicalized and unlexicalized trees; (ii) a question classification dataset; (iii) question and answer pairs from Yahoo! answers; and (vi) a new dataset from INEX [39].

We evaluated the speedup in terms of the training time and the number of Δ-iterations for both STK and the newly proposed PTK for SDAG and

SDAG+ on the above datasets. The results have shown that: (1) our approach generalizes to most of tree-based kernels as we obtain significant speedup of PTK-based learning; and (2) when the training data is relatively small (up[13] to 100k) the compactness of the SDAG+ models allows for better usage of the CPU cache, amplifying the benefit of our approach; (3) the results on the NLP tasks underrepresent the potential of our approach as the subtrees are based on words, which make subtrees sparser. Indeed, the results on INEX show speedup up to 283 in terms of Δ computations and up to 77 in runtime.

Our study opens several future research directions: application of tree kernels to many tasks, where large data size has prevented their use. This surely regards SRL in many languages but also parse tree re-ranking [8] and question answering applications. Also applications to other data mining tasks would be interesting, e.g., XML tree classification.

From the algorithmic perspective, it would be promising to explore approaches to prune the DAGs for achieving higher compression rates without any loss in accuracy. Finally, the ultimate goal would be to use tree kernels for structured output prediction.

Acknowledgements This work has been partially supported by the EC project FP247758: Trustworthy Eternal Systems via Evolving Software, Data and Knowledge (EternalS).

References

1. Aiolli, F., Martino, G.D.S., Sperduti, A., Moschitti, A.: Fast on-line kernel learning for trees. In: ICDM, pp. 787–791 (2006)
2. Aiolli, F., Martino, G.D.S., Sperduti, A., Moschitti, A.: Efficient kernel-based learning for trees. In: CIDM, pp. 308–315 (2007)
3. Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S.: Efficient substructure discovery from large semi-structured data. In: SDM (2002)
4. Cancedda, N., Gaussier, E., Goutte, C., Renders, J.M.: Word sequence kernels. Journal of Machine Learning Research **3**, 1059–1082 (2003)
5. Carreras, X., Màrquez, L.: Introduction to the CoNLL-2005 Shared Task: Semantic Role Labeling. In: Proceedings of the 9th Conference on Natural Language Learning, CoNLL-2005. Ann Arbor, MI USA (2005)
6. Charniak, E.: A maximum-entropy-inspired parser. In: ANLP, pp. 132–139 (2000)
7. Chi, Y., Yang, Y., Muntz, R.R.: Hybridtreeminer: An efficient algorithm for mining frequent rooted trees and free trees using canonical form. In: SSDBM, pp. 11–20 (2004)
8. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In: ACL, pp. 263–270 (2002)
9. Cumby, C., Roth, D.: Kernel Methods for Relational Learning. In: Proceedings of ICML 2003 (2003)
10. Daumé III, H., Marcu, D.: A tree-position kernel for document compression. In: Proceedings of the DUC. Boston, MA (2004)
11. Denoyer, L., Gallinari, P.: Report on the xml mining track at inex 2005 and inex 2006: categorization and clustering of xml documents. SIGIR Forum **41**, 79–90 (2007)
12. Franc, V., Sonnenburg, S.: Optimized cutting plane algorithm for support vector machines. In: ICML, pp. 320–327 (2008)
13. Haussler, D.: Convolution kernels on discrete structures. Tech. Rep. UCSC-CRL-99-10, University of California, Santa Cruz (1999)

[13] Of course, this mainly depends on the hardware characteristics.

14. Joachims, T.: Making large-scale SVM learning practical. In: Advances in Kernel Methods - Support Vector Learning, chap. 11, pp. 169–184. MIT Press, Cambridge, MA (1999)
15. Joachims, T.: A support vector method for multivariate performance measures. In: International Conference on Machine Learning (ICML), pp. 377–384 (2005)
16. Joachims, T.: Training linear SVMs in linear time. In: KDD (2006)
17. Joachims, T., Yu, C.N.J.: Sparse kernel svms via cutting-plane training. Machine Learning **76**(2-3), 179–193 (2009). ECML
18. Kate, R.J., Mooney, R.J.: Using string-kernels for learning semantic parsers. In: ACL (2006)
19. Kuang, R., Ie, E., Wang, K., Wang, K., Siddiqi, M., Freund, Y., Leslie, C.S.: Profile-based string kernels for remote homology detection and motif extraction. In: 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference (CSB 2004), pp. 152–160 (2004)
20. Kudo, T., Matsumoto, Y.: Fast methods for kernel-based text analysis. In: Proceedings of ACL'03 (2003)
21. Leslie, C., Eskin, E., Cohen, A., Weston, J., Noble, W.S.: Mismatch string kernels for discriminative protein classification. Bioinformatics **20**(4), 467–76 (2004)
22. Marcus, M., Santorini, B., Marcinkiewicz, M.: Building a large annotated corpus of English: the Penn Treebank. Computational Linguistics **19**(2), 313–330 (1993)
23. Moschitti, A.: Efficient convolution kernels for dependency and constituent syntactic trees. In: Proceedings of ECML (2006)
24. Moschitti, A.: Making tree kernels practical for natural language learning. In: EACL. The Association for Computer Linguistics (2006)
25. Moschitti, A., Pighin, D., Basili, R.: Tree kernels for semantic role labeling. Computational Linguistics **34**(2), 193–224 (2008)
26. Noreen, E.W.: Computer-Intensive Methods for Testing Hypotheses : An Introduction. Wiley-Interscience (1989)
27. Padó, S.: User's guide to `sigf`: Significance testing by approximate randomisation (2006)
28. Palmer, M., Kingsbury, P., Gildea, D.: The proposition bank: An annotated corpus of semantic roles. Computational Linguistics **31**(1), 71–106 (2005)
29. Pighin, D., Moschitti, A.: Efficient linearization of tree kernel functions. In: Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL-2009), pp. 30–38. Association for Computational Linguistics, Boulder, Colorado (2009)
30. Saigo, H., Vert, J., Akutsu, T., Ueda, N.: Protein homology detection using string alignment kernels. Bioinformatics **20**, 1682–1689 (2004)
31. Severyn, A., Moschitti, A.: Large-scale support vector learning with structural kernels. In: ECML/PKDD (3), pp. 229–244 (2010)
32. Severyn, A., Moschitti, A.: Fast support vector machines for structural kernels. In: ECML (2011)
33. Shasha, D., Wang, J.T.L., Zhang, S.: Unordered tree mining with applications to phylogeny. In: ICDE, pp. 708–719 (2004)
34. Shen, L., Sarkar, A., Joshi, A.k.: Using LTAG Based Features in Parse Reranking. In: Proceedings of EMNLP'06 (2003)
35. Shervashidze, N., Borgwardt, K.: Fast subtree kernels on graphs. Proceedings of Advances in Neural Information Processing Systems (2009)
36. Shi, Q., Petterson, J., Dror, G., Langford, J., Smola, A.J., Vishwanathan, S.V.N.: Hash kernels for structured data. JMLR **10**, 2615–2637 (2009)
37. Steinwart, I.: Sparseness of support vector machines. Journal of Machine Learning Research **4**, 1071–1105 (2003)
38. Termier, A., Rousset, M.C., Sebag, M.: Dryade: A new approach for discovering closed frequent trees in heterogeneous tree databases. In: ICDM, pp. 543–546 (2004)
39. Trentini, F., Hagenbuchner, M., Sperduti, A., Scarselli, F.: A self-organising map approach for clustering of xml documents. In: IJCNN, pp. 1805–1812. IEEE (2006)
40. Tsochantaridis, I., Joachims, T., Hofmann, T., Altun, Y.: Large margin methods for structured and interdependent output variables. Journal of Machine Learning Research **6**, 1453–1484 (2005)
41. Veropoulos, K., Campbell, C., Cristianini, N.: Controlling the sensitivity of support vector machines. In: Proceedings of the IJCAI, pp. 55–60 (1999)

42. Wang, C., Hong, M., Pei, J., Zhou, H., Wang, W., Shi, B.: Efficient pattern-growth methods for frequent tree pattern mining. In: PAKDD, pp. 441–451 (2004)
43. Wu, G., Chang, E.: Class-boundary alignment for imbalanced dataset learning. ICML 2003 workshop on learning from imbalanced data sets II, Washington, DC pp. 49–56 (2003)
44. Xia, Y., Yang, Y.: Mining closed and maximal frequent subtrees from databases of labeled rooted trees. IEEE Transactions on Knowledge and Data Engineering **17**(2), 190–202 (2005). Student Member-Yun Chi and Fellow-Richard R. Muntz
45. Yang, L.H., Lee, M.L., Hsu, W., Guo, X.: 2pxminer: an efficient two pass mining of frequent xml query patterns. In: KDD, pp. 731–736 (2004)
46. Yu, C.N.J., Joachims, T.: Training structural svms with kernels using sampled cuts. In: KDD, pp. 794–802 (2008)
47. Zadrozny, B., Langford, J., Abe, N.: Cost-sensitive learning by cost-proportionate example weighting. In: Proceedings of ICDM (2003)
48. Zaki, M.J.: Efficiently mining frequent trees in a forest: Algorithms and applications. Knowledge and Data Engineering, IEEE Transactions on **17**(8), 1021–1035 (2005)