Efficient Linearization of Tree Kernel Functions

Daniele Pighin
FBK-Irst, HLT
Via di Sommarive, 18 I-38100 Povo (TN) Italy
pighin@fbk.eu

Alessandro Moschitti
University of Trento, DISI
Via di Sommarive, 14 I-38100 Povo (TN) Italy
moschitti@disi.unitn.it

Abstract

The combination of Support Vector Machines with very high dimensional kernels, such as string or tree kernels, suffers from two major drawbacks: first, the implicit representation of feature spaces does not allow us to understand which features actually triggered the generalization; second, the resulting computational burden may in some cases render unfeasible to use large data sets for training. We propose an approach based on feature space reverse engineering to tackle both problems. Our experiments with Tree Kernels on a Semantic Role Labeling data set show that the proposed approach can drastically reduce the computational footprint while yielding almost unaffected accuracy.

1 Introduction

The use of Support Vector Machines (SVMs) in supervised learning frameworks is spreading across different communities, including Computational Linguistics and Natural Language Processing, thanks to their solid mathematical foundations, efficiency and accuracy. Another important reason for their success is the possibility of using kernel functions to implicitly represent examples in some high dimensional kernel space, where their similarity is evaluated. Kernel functions can generate a very large number of features, which are then weighted by the SVM optimization algorithm obtaining a feature selection side-effect. Indeed, the weights encoded by the gradient of the separating hyperplane learnt by the SVM implicitly establish a ranking between features in the kernel space. This property has been exploited in feature selection models based on approximations or transformations of the gradient, e.g. (Rakotomamonjy, 2003), (Weston et al., 2003) or (Kudo and Matsumoto, 2003).

However, kernel based systems have two major drawbacks: first, new features may be discovered in the implicit space but they cannot be directly observed. Second, since learning is carried out in the dual space, it is not possible to use the faster SVM or perceptron algorithms optimized for linear spaces. Consequently, the processing of large data sets can be computationally very expensive, limiting the use of large amounts of data for our research or applications.

We propose an approach that tries to fill in the gap between explicit and implicit feature representations by 1) selecting the most relevant features in accordance with the weights estimated by the SVM and 2) using these features to build an explicit representation of the kernel space. The most innovative aspect of our work is the attempt to model and implement a solution in the context of structural kernels. In particular we focus on Tree Kernel (TK) functions, which are especially interesting for the Computational Linguistics community as they can effectively encode rich syntactic data into a kernel-based learning algorithm. The high dimensionality of a TK feature space poses interesting challenges in terms of computational complexity that we need to address in order to come up with a viable solution. We will present a number of experiments carried out in the context of Semantic Role Labeling, showing that our approach can noticeably reduce training time while yielding almost unaffected classification accuracy, thus allowing us to handle larger data sets at a reasonable computational cost.

The rest of the paper is structured as follows: Sec-

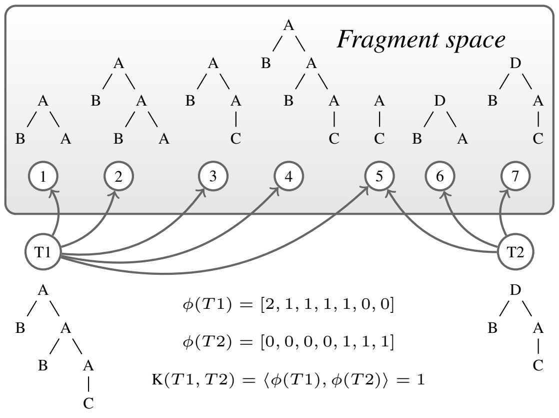
Fragment space
T1
T2
$\phi(T1) = [2,1,1,1,1,0,0]$
$\phi(T2) = [0,0,0,0,1,1,1]$
$K(T1,T2) = \langle\phi(T1),\phi(T2)\rangle = 1$

Figure 1: Esemplification of a fragment space and the kernel product between two trees.

tion 2 will briefly review SVMs and Tree Kernel functions; Section 3 will detail our proposal for the linearization of a TK feature space; Section 4 will review previous work on related subjects; Section 5 will describe our experiments and comment on their results; finally, in Section 6 we will draw our conclusions.

2 Tree Kernel Functions

The decision function of an SVM is:

$$f(\vec{x}) = \vec{w} \cdot \vec{x} + b = \sum_{i=1}^{n} \alpha_i y_i \vec{x_i} \cdot \vec{x} + b \qquad (1)$$

where $\vec{x}$ is a classifying example and $\vec{w}$ and b are the separating hyperplane's *gradient* and its *bias*, respectively. The gradient is a linear combination of the training points $\vec{x_i}$, their labels y_i and their weights α_i. These and the bias are optimized at training time by the learning algorithm. Applying the so-called *kernel trick* it is possible to replace the scalar product with a *kernel function* defined over pairs of *objects*:

$$f(o) = \sum_{i=1}^{n} \alpha_i y_i \mathrm{k}(o_i, o) + b$$

with the advantage that we do not need to provide an explicit mapping $\phi(\cdot)$ of our examples in a vector space.

A Tree Kernel function is a convolution kernel (Haussler, 1999) defined over pairs of trees. Practically speaking, the kernel between two trees evaluates the number of substructures (or *fragments*) they have in common, i.e. it is a measure of their overlap. The function can be computed recursively in closed form, and quite efficient implementations are available (Moschitti, 2006). Different TK functions are characterized by alternative fragment definitions, e.g. (Collins and Duffy, 2002) and (Kashima and Koyanagi, 2002). In the context of this paper we will be focusing on the SubSet Tree (SST) kernel described in (Collins and Duffy, 2002), which relies on a fragment definition that does not allow to break production rules (i.e. if any child of a node is included in a fragment, then also all the other children have to). As such, it is especially indicated for tasks involving constituency parsed texts.

Implicitly, a TK function establishes a correspondence between distinct fragments and dimensions in some *fragment space*, i.e. the space of all the possible fragments. To simplify, a tree t can be represented as a vector whose attributes count the occurrences of each fragment within the tree. The kernel between two trees is then equivalent to the scalar product between pairs of such vectors, as exemplified in Figure 1.

3 Mining the Fragment Space

If we were able to efficiently mine and store in a dictionary all the fragments encoded in a model, we would be able to represent our objects explicitly and use these representations to train larger models and very quick and accurate classifiers. What we need to devise are strategies to make this approach convenient in terms of computational requirements, while yielding an accuracy comparable with direct tree kernel usage.

Our framework defines five distinct activities, which are detailed in the following paragraphs.

Fragment Space Learning (*FSL*) First of all, we can partition our training data into S smaller sets, and use the SVM and the SST kernel to learn S models. We will use the estimated weights to drive our feature selection process. Since the time complexity of SVM training is approximately quadratic in the number of examples, this way we can considerably accelerate the process of estimating support vector weights.

According to statistical learning theory, being trained on smaller subsets of the available data these models will be less robust with respect to the minimization of the empirical risk (Vapnik, 1998). Nonetheless, since we do not need to employ them for classification (but just to direct our feature selection process, as we will describe shortly), we can accept to rely on sub-optimal weights. Furthermore, research results in the field of SVM parallelization using cascades of SVMs (Graf et al., 2004) suggest that support vectors collected from locally learnt models can encode many of the relevant features retained by models learnt globally. Henceforth, let M_s be the model associated with the s-th split, and $\mathcal{F}_s$ the fragment space that can describe all the trees in M_s.

Fragment Mining and Indexing (*FMI*) In Equation 1 it is possible to isolate the gradient $\vec{w} = \sum_{i=1}^{n} \alpha_i y_i \vec{x_i}$, with $\vec{x_i} = [x_i^{(1)}, \ldots, x_i^{(N)}]$, N being the dimensionality of the feature space. For a tree kernel function, we can rewrite $x_i^{(j)}$ as:

$$x_i^{(j)} = \frac{t_{i,j}\lambda^{\ell(f_j)}}{\|t_i\|} = \frac{t_{i,j}\lambda^{\ell(f_j)}}{\sqrt{\sum_{k=1}^{N}(t_{i,k}\lambda^{\ell(f_k)})^2}} \qquad (2)$$

where: $t_{i,j}$ is the number of occurrences of the fragment f_j, associated with the j-th dimension of the feature space, in the tree t_i; λ is the kernel decay factor; and $\ell(f_j)$ is the depth of the fragment.

The relevance $|w^{(j)}|$ of the fragment f_j can be measured as:

$$|w^{(j)}| = \left|\sum_{i=1}^{n} \alpha_i y_i x_i^{(j)}\right| . \qquad (3)$$

We fix a threshold L and from each model M_s (learnt during FSL) we select the L most relevant fragments, i.e. we build the set $\mathcal{F}_{s,L} = \cup_k \{f_k\}$ so that:

$$|\mathcal{F}_{s,L}| = L \text{ and } |w^{(k)}| \geq |w^{(i)}| \forall f_i \in \mathcal{F} \setminus \mathcal{F}_{s,L} .$$

In order to do so, we need to harvest all the fragments with a fast extraction function, store them in a compact data structure and finally select the fragments with the highest relevance. Our strategy is exemplified in Figure 2. First, we represent each fragment as a sequence as described in (Zaki, 2002). A sequence contains the labels of the fragment nodes in depth-first order. By default, each node is the child of the previous node in the sequence. A special symbol ($\uparrow$) indicates that the next node in the sequence should be attached after climbing one level in the tree. For example, the tree *(B (Z W))* in figure is represented as the sequence *[B, Z, $\uparrow$, W]*. Then, we add the elements of the sequence to a graph (which we call an *index* of fragments) where each sequence becomes a path. The nodes of the index are the labels of the fragment nodes, and each arc is associated with a pair of values $\langle d, n\rangle$: d is a node identifier, which is unique with respect to the source node; n is the identifier of the arc that must be selected at the destination node in order to follow the path associated with the sequence. Index nodes associated with a fragment root also have a field where the cumulative relevance of the fragment is stored.

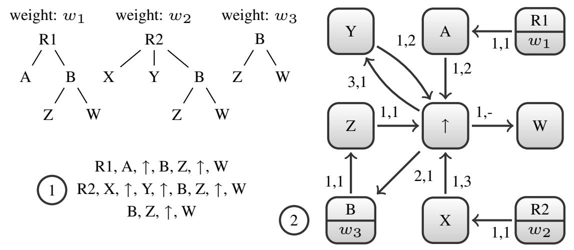

Figure 2: Fragment indexing. Each fragment is represented as a sequence ① and then encoded as a path in the index ② which keeps track of its cumulative relevance.

As an example, the index node labeled *B* in figure has an associated weight of w_3, thus identifying the root of a fragment. Each outgoing edge univocally identifies an indexed fragment. In this case, the only outgoing edge is labeled with the pair $\langle d = 1, n = 1\rangle$, meaning that we should follow it to the next node, i.e. *Z*, and there select the edge labeled *1*, as indicated by n. The edge with $d = 1$ in *Z* is $\langle d = 1, n = 1\rangle$, so we browse to $\uparrow$ where we select the edge $\langle d = 1, n = -\rangle$. The missing value for n tells us that the next node, *W*, is the last element of the sequence. The complete sequence is then *[B, Z, $\uparrow$, W]*, which encodes the fragment *(B (Z W))*.

The index implementation has been optimized for fast insertions and has the following features: 1) each node label is represented exactly once; 2) each distinct sequence tail is represented exactly once. The union of all the fragments harvested from each model is then saved into a dictionary $\mathcal{D}_L$ which will be used by the next stage.

To mine the fragments, we apply to each tree in each model the algorithm shown in Algorithm 3.1. In this context, we call *fragment expansion* the pro-

Algorithm 3.1: MINE_TREE($tree$)

```
global $maxdepth, maxexp$
main
  $mined \leftarrow \emptyset; indexed \leftarrow \emptyset;$ MINE(FRAG($tree$), 0)
procedure MINE($frag, depth$)
  if $frag \in indexed$
    then return
  $indexed \leftarrow indexed \cup \{frag\}$
  INDEX($frag$)
  for each $node \in$ TO_EXPAND($frag$)
    do { if $node \notin mined$
         then { $mined \leftarrow mined \cup \{node\}$
                MINE(FRAG($node$), 0)
  if $depth < maxdepth$
    then { for each $fragment \in$ EXPAND($frag, maxexp$)
             do MINE($fragment, depth + 1$)
```

cess by which tree nodes are included in a fragment. Fragment expansion is achieved via *node expansions*, where expanding a node means including its direct children in the fragment. The function FRAG(n) builds the basic fragment rooted in a given node n, i.e. the fragment consisting only of n and its direct children. The function TO_EXPAND(f) returns the set of nodes in a fragment f that can be expanded (i.e. internal nodes in the origin tree), while the function EXPAND($f, maxexp$) returns all the possible expansions of a fragment f. The parameter $maxexp$ is a limit to the number of nodes that can be expanded at the same time when a new fragment is generated, while $maxdepth$ sets a limit on the number of times that a base fragment can be expanded. The function INDEX(f) adds the fragment f to the index. To keep the notation simple, here we assume that a fragment f contains all the necessary information to calculate its relevance (i.e. the weight, label and norm of the support vector α_i, y_i, and $\|t_i\|$, the depth of the fragment $\ell(f)$ and the decay factor λ, see equations 2 and 3).

Performing in a different order the same node expansions on the same fragment f results in the same fragment f'. To prevent the algorithm from entering circular loops, we use the set $indexed$ so that the very same fragment in each tree cannot be explored more than once. Similarly, the $mined$ set is used so that the base fragment rooted in a given node is considered only once.

Tree Fragment Extraction (*TFX*) During this phase, a data file encoding label-tree pairs $\langle y_i, t_i \rangle$ is transformed to encode label-vector pairs $\langle y_i, \vec{v_i} \rangle$. To do so, we generate the fragment space of t_i, using a variant of the mining algorithm described in Figure 3.1, and encode in $\vec{v_i}$ all and only the fragments $t_{i,j}$ so that $t_{i,j} \in \mathcal{D}_L$, i.e. we perform feature extraction based on the indexed fragments. The process is applied to the whole training and test sets. The algorithm exploits labels and production rules found in the fragments listed in the dictionary to generate only the fragments that *may be* in the dictionary. For example, if the dictionary does not contain a fragment whose root is labeled N, then if a node N is encountered during TFX neither its base fragment nor its expansions are generated.

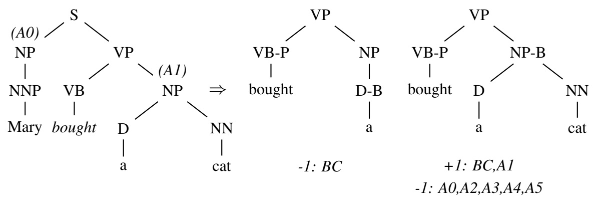

Figure 3: Examples of AST$_m$ structured features.

Explicit Space Learning (*ESL*) After linearizing the training data, we can learn a very fast model by using all the available data and a linear kernel. The fragment space is now *explicit*, as there is a mapping between the input vectors and the fragments they encode.

Explicit Space Classification (*ESC*) After learning the linear model, we can classify our linearized test data and evaluate the accuracy of the resulting classifier.

4 Previous work

A rather comprehensive overview of feature selection techniques is carried out in (Guyon and Elisseeff, 2003). Non-filter approaches for SVMs and kernel machines are often concerned with polynomial and Gaussian kernels, e.g. (Weston et al., 2001) and (Neumann et al., 2005). Weston et al. (2003) use the ℓ_0 norm in the SVM optimizer to stress the feature selection capabilities of the learning algorithm. In (Kudo and Matsumoto, 2003), an extension of the PrefixSpan algorithm (Pei et al., 2001) is used to efficiently mine the features in a low degree polynomial kernel space. The authors discuss an approximation of their method that allows them to handle high degree polynomial kernels.

Data set			**Non-linearized classifiers**					**Linearized classifiers (Thr=10k)**				
Task	Pos	Neg	Train	Test	P	R	F_1	Train	Test	P	R	F_1
A0	60,900	118,191	521	7	90.26	92.95	91.59	209	3	88.95	91.91	90.40
A1	90,636	88,455	1,206	11	89.45	88.62	89.03	376	3	89.39	88.13	88.76
A2	21,291	157,800	692	7	84.56	64.42	73.13	248	3	81.23	68.29	74.20
A3	3,481	175,610	127	2	97.67	40.00	56.76	114	3	97.56	38.10	54.79
A4	2,713	176,378	47	1	92.68	55.07	69.10	92	2	95.00	55.07	69.72
A5	69	179,022	3	0	100.00	50.00	66.67	63	2	100.00	50.00	66.67
BC	61,062	938,938	3,059	247	82.57	80.96	81.76	916	39	83.36	78.95	81.10
RM	-	-	2,596	27	89.37	86.00	87.65	1,090	16	88.50	85.81	87.13

Table 1: Accuracy (*P*, *R*, F_1), training (*Train*) and test (*Test*) time of non-linearized (center) and linearized (right) classifiers. Times are in minutes. For each task, columns *Pos* and *Neg* list the number of positive and negative training examples, respectively. The accuracy of the role multiclassifiers is the micro-average of the individual classifiers trained to recognize core PropBank roles.

Suzuki and Isozaki (2005) present an embedded approach to feature selection for convolution kernels based on χ^2-driven relevance assessment. To our knowledge, this is the only published work clearly focusing on feature selection for tree kernel functions. In (Graf et al., 2004), an approach to SVM parallelization is presented which is based on a divide-et-impera strategy to reduce optimization time. The idea of using a compact graph representation to represent the support vectors of a TK function is explored in (Aiolli et al., 2006), where a Direct Acyclic Graph (DAG) is employed.

Concerning the use of kernels for NLP, interesting models and results are described, for example, in (Collins and Duffy, 2002), (Moschitti et al., 2008), (Kudo and Matsumoto, 2003), (Cumby and Roth, 2003), (Shen et al., 2003), (Cancedda et al., 2003), (Culotta and Sorensen, 2004), (Daumé III and Marcu, 2004), (Kazama and Torisawa, 2005), (Kudo et al., 2005), (Titov and Henderson, 2006), (Moschitti et al., 2006), (Moschitti and Bejan, 2004) or (Toutanova et al., 2004).

5 Experiments

We tested our model on a Semantic Role Labeling (SRL) benchmark, using PropBank annotations (Palmer et al., 2005) and automatic Charniak parse trees (Charniak, 2000) as provided for the CoNLL 2005 evaluation campaign (Carreras and Màrquez, 2005). SRL can be decomposed into two tasks: *boundary detection*, where the word sequences that are arguments of a predicate word w are identified, and *role classification*, where each argument is assigned the proper role. The former task requires a binary *Boundary Classifier* (BC), whereas the second involves a *Role Multi-class Classifier* (RM).

Setup. If the constituency parse tree t of a sentence s is available, we can look at all the pairs $\langle p, n_i \rangle$, where n_i is any node in the tree and p is the node dominating w, and decide whether n_i is an *argument node* or not, i.e. whether it exactly dominates all and only the words encoding any of w's arguments. The objects that we classify are subsets of the input parse tree that encompass both p and n_i. Namely, we use the AST$_m$ structure defined in (Moschitti et al., 2008), which is the minimal tree that covers all and only the words of p and n_i. In the AST$_m$, p and n_i are marked so that they can be distinguished from the other nodes. An AST$_m$ is regarded as a positive example for BC if n_i is an argument node, otherwise it is considered a negative example. Positive BC examples can be used to train an efficient RM: for each role r we can train a classifier whose positive examples are argument nodes whose label is exactly r, whereas negative examples are argument nodes labeled $r' \neq r$. Two AST$_m$s extracted from an example parse tree are shown in Figure 3: the first structure is a negative example for BC and is not part of the data set of RM, whereas the second is a positive instance for BC and A1.

To train BC we used PropBank sections 1 through 6, extracting AST$_m$ structures out of the first 1 million $\langle p, n_i \rangle$ pairs from the corresponding parse trees. As a test set we used the 149,140 instance collected from the annotations in Section 24. There are 61,062 positive examples in the training set (i.e. 6.1%) and 8,515 in the test set (i.e. 5.7%).

For RM we considered all the argument nodes of any of the six PropBank core roles (i.e. A0, ...,

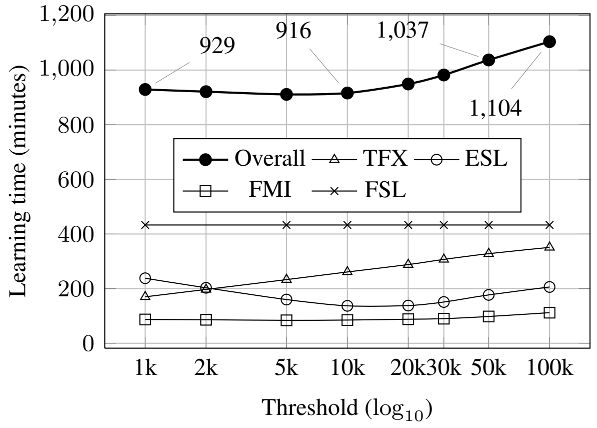

Figure 4: Training time decomposition for the linearized BC with respect to its main components when varying the threshold value.

A5) from all the available training sections, i.e. 2 through 21, for a total of 179,091 training instances. Similarly, we collected 5,928 test instances from the annotations of Section 24.

In the remainder, we will mark with an ℓ the linearized classifiers, i.e. BC_ℓ and RM_ℓ will refer to the linearized boundary and role classifiers, respectively. Their traditional, vanilla SST counterparts will be simply referred to as BC and RM.

We used 10 splits for the FMI stage and we set $maxdepth = 4$ and $maxexp = 5$ during FMI and TFX. We didn't carry out an extensive validation of these parameters. These values were selected during the development of the software because, on a very small development set, they resulted in a very responsive system.

Since the main topic of this paper is the assessment of the efficiency and accuracy of our linearization technique, we did not carry out an evaluation on the whole SRL task using the official CoNLL'05 evaluator. Indeed, producing complete annotations requires several steps (e.g. overlap resolution, OvA or Pairwise combination of individual role classifiers) that would shade off the actual impact of the methodology on classification.

Platform. All the experiments were run on a machine equipped with 4 Intel® Xeon® CPUs clocked at 1.6 GHz and 4 GB of RAM running on a Linux 2.6.9 kernel. As a supervised learning framework we used SVM-Light-TK [1], which extends the SVM-Light optimizer (Joachims, 2000) with tree kernel support. During FSL, we learn the models using a normalized SST kernel and the default decay factor $\lambda = 0.4$. The same parameters are used to train the models of the non linearized classifiers. During ESL, the classifier is trained using a linear kernel. We did not carry out further parametrization of the learning algorithm.

[1] http://disi.unitn.it/~moschitt/Tree-Kernel.htm

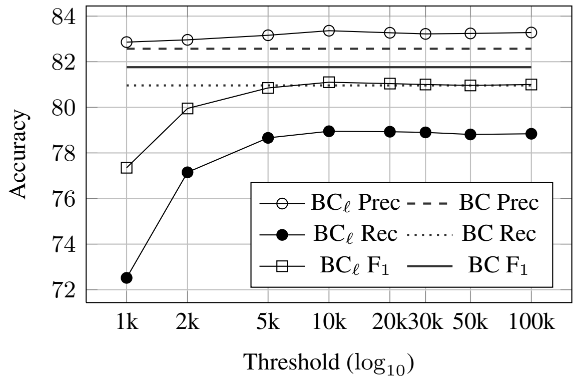

Figure 5: BC_ℓ accuracy for different thresholds.

Results. The left side of Table 1 shows the distribution of positive (Column *Pos*) and negative (*Neg*) data points in each classifier's training set. The central group of columns lists training and test efficiency and accuracy of BC and RM, i.e. the non-linearized classifiers, along with figures for the individual role classifiers that make up RM.

Training BC took more than two days of CPU time and testing about 4 hours. The classifier achieves an F_1 measure of 81.76, with a good balance between precision and recall. Concerning RM, sequential training of the 6 models took 2,596 minutes, while classification took 27 minutes. The slowest of the individual role classifiers happens to be A1, which has an almost 1:1 ratio between positive and negative examples, i.e. they are 90,636 and 88,455 respectively.

We varied the threshold value (i.e. the number of fragments that we mine from each model, see Section 3) to measure its effect on the resulting classifier accuracy and efficiency. In this context, we call *training time* all the time necessary to obtain a linearized model, i.e. the sum of FSL, FMI and TFX time for every split, plus the time for ESL. Similarly, we call *test time* the time necessary to classify a linearized test set, i.e. the sum of TFX and ESC on test data.

In Figure 4 we plot the efficiency of BC_ℓ learning with respect to different threshold values. The Overall training time is shown alongside with partial times coming from FSL (which is the same for every threshold value and amounts to 433 minutes), FMI, training data TFX and ESL. The plot shows that TFX has a logarithmic behaviour, and that quite soon becomes the main player in total training time after FSL. For threshold values lower than 10k, ESL time decreases as the threshold increases: too few fragments are available and adding new ones increases the probability of including relevant fragments in the dictionary. After 10k, all the relevant fragments are already there and adding more only makes computation harder. We can see that for a threshold value of 100k total training time amounts to 1,104 minutes, i.e. 36% of BC. For a threshold value of 10k, learning time further decreases to 916 minutes, i.e. less than 30%. This threshold value was used to train the individual linearized role classifiers that make up RM_ℓ.

These considerations are backed by the trend of classification accuracy shown in Figure 5, where the Precision, Recall and F_1 measure of BC_ℓ, evaluated on the test set, are shown in comparison with BC. We can see that BC_ℓ precision is almost constant, while its recall increases as we increase the threshold, reaches a maximum of 78.95% for a threshold of 10k and then settles around 78.8%. The F_1 score is maximized for a threshold of 10k, where it measures 81.10, i.e. just 0.66 points less than BC. We can also see that BC_ℓ is constantly more conservative than BC, i.e. it always has higher precision and lower recall.

Table 1 compares side to side the accuracy (columns P, R and F_1), training (*Train*) and test (*Test*) times of the different classifiers (central block of columns) and their linearized counterparts (block on the right). Times are measured in minutes. For the linearized classifiers, test time is the sum of TFX and ESC time, but the only relevant contribution comes from TFX, as the low dimensional linear space and fast linear kernel allow us to classify test instances very efficiently [2]. Overall, BC_ℓ test time is 39 minutes, which is more than 6 times faster than BC (i.e. 247 minutes). It should be stressed that we are comparing against a fast TK implementation that is almost linear in time with respect to the number of tree nodes (Moschitti, 2006).

[2] Although ESC is not shown in table, the classification of all 149k test instances with BC_ℓ took 5 seconds with a threshold of 1k and 17 seconds with a threshold of 100k.

Learning parallelization

Task	Non Lin.	Linearized (Thr=10k)		
		1 *cpu*	5 *cpu*s	10 *cpu*s
BC	3,059	916	293	215
RM	2,596	1,090	297	198

Table 2: Learning time when exploiting the framework's parallelization capabilities. Column *Non Lin.* lists non-linearized training time.

Concerning RM_ℓ, we can see that the accuracy loss is even less than with BC_ℓ, i.e. it reaches an F_1 measure of 87.13 which is just 0.52 less than RM. It is also interesting to note how the individual linearized role classifiers manage to perform accurately regardless of the distribution of examples in the data set: for all the six classifiers the final accuracy is in line with that of the corresponding non-linearized classifier. In two cases, i.e. A2 and A4, the accuracy of the linearized classifier is even higher, i.e. 74.20 vs. 73.13 and 69.72 vs. 69.10, respectively. As for the efficiency, total training time for RM_ℓ is 37% of RM, i.e. 1,190 vs. 2,596 minutes, while test time is reduced to 60%, i.e. 16 vs 27 minutes. These improvements are less evident than those measured for boundary detection. The main reason is that the training set for boundary classification is much larger, i.e. 1 million vs. 179k instances: therefore, splitting training data during FSL has a reduced impact on the overall efficiency of RM_ℓ.

Parallelization. All the efficiency improvements that have been discussed so far considered a completely sequential process. But one of the advantages of our approach is that it allows us to parallelize some aspect of SVM training. Indeed, every activity (but ESL) can exploit some degree of parallelism: during FSL, all the models can be learnt at the same time (for this activity, the maximum degree of parallelization is conditioned by the number of training data splits); during FMI, models can be mined concurrently; during TFX, the data-set to be linearized can be split arbitrarily and individual segments can be processed in parallel. Exploiting this possibility we can drastically improve learning efficiency. As an example, in Table 2 we show how the total learning of the BC_ℓ can be cut to as low as 215 seconds when exploiting ten CPUs and using a

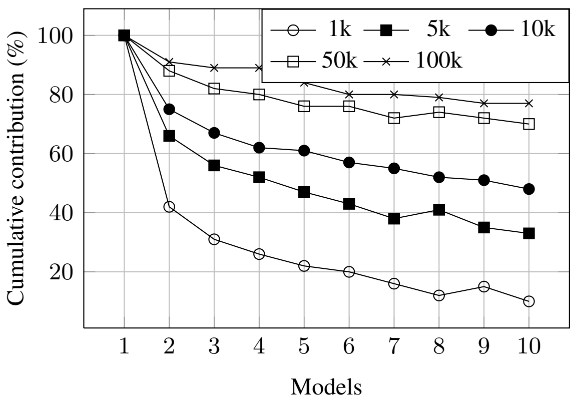

Figure 6: Growth of dictionary size when including fragments from more splits at different threshold values. When a low threshold is used, the contribution of individual dictionaries tends to be more marginal.

threshold of 10k. Even running on just 5 CPUs, the overall computational cost of BC_ℓ is less than 10% of BC (Column *Non Lin.*). Similar considerations can be drawn concerning the role multi-classifier.

Fragment space. In this section we take a look at the fragments included in the dictionary of the BC_ℓ classifier. During FMI, we incrementally merge the fragments mined from each of the models learnt during FSL. Figure 6 plots, for different threshold values, the percentage of new fragments (on the y axis) that the i-th model (on the x axis) contributes with respect to the number of fragments mined from each model (i.e. the threshold value).

If we consider the curve for a threshold equal to 100k, we can see that each model after the first approximately contributes with the same number of fragments. On the other hand, if the threshold is set to 1k than the contribution of subsequent models is increasingly more marginal. Eventually, less than 10% of the fragments mined from the last model are new ones. This behaviour suggests that there is a core set of very relevant fragments which is common across models learnt on different data, i.e. they are relevant for the task and do not strictly depend on the training data that we use. When we increase the threshold value, the new fragments that we index are more and more data specific.

The dictionary compiled with a threshold of 10k lists 62,760 distinct fragments. 15% of the fragments contain the predicate node (which generally is the node encoding the predicate word's POS tag), more than one third contain the candidate argument node and, of these, about one third are rooted in it. This last figure strongly suggests that the internal structure of an argument is indeed a very powerful feature not only for role classification, as we would expect, but also for boundary detection. About 10% of the fragments contain both the predicate and the argument node, while about 1% encode the Path feature traditionally used in explicit semantic role labeling models (Gildea and Jurafsky, 2002). About 5% encode a sort of extended Path feature, where the argument node is represented together with its descendants. Overall, about 2/3 of the fragments contain at least some terminal symbol (i.e. words), generally a preposition or an adverb.

6 Conclusions

We presented a supervised learning framework for Support Vector Machines that tries to combine the power and modeling simplicity of convolution kernels with the advantages of linear kernels and explicit feature representations. We tested our model on a Semantic Role Labeling benchmark and obtained very promising results in terms of accuracy and efficiency. Indeed, our linearized classifiers manage to be almost as accurate as non linearized ones, while drastically reducing the time required to train and test a model on the same amounts of data.

To our best knowledge, the main points of novelty of this work are the following: 1) it addresses the problem of feature selection for tree kernels, exploiting SVM decisions to guide the process; 2) it provides an effective way to make the kernel space observable; 3) it can efficiently linearize structured data without the need for an explicit mapping; 4) it combines feature selection and SVM parallelization.

We began investigating the fragments generated by a TK function for SRL, and believe that studying them in more depth will be useful to identify new, relevant features for the characterization of predicate-argument relations.

In the months to come, we plan to run a set of experiments on a wider list of tasks so as to consolidate the results we obtained so far. We will also test the generality of the approach by testing with different high-dimensional kernel families, such as sequence and polynomial kernels.

References

Fabio Aiolli, Giovanni Da San Martino, Alessandro Sperduti, and Alessandro Moschitti. 2006. Fast on-line kernel learning for trees. In *Proceedings of ICDM'06*.

Nicola Cancedda, Eric Gaussier, Cyril Goutte, and Jean Michel Renders. 2003. Word sequence kernels. *Journal of Machine Learning Research*, 3:1059–1082.

Xavier Carreras and Lluís Màrquez. 2005. Introduction to the CoNLL-2005 Shared Task: Semantic Role Labeling. In *Proceedings of CoNLL'05*.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings of NAACL'00*.

Michael Collins and Nigel Duffy. 2002. New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron. In *Proceedings of ACL'02*.

Aron Culotta and Jeffrey Sorensen. 2004. Dependency Tree Kernels for Relation Extraction. In *Proceedings of ACL'04*.

Chad Cumby and Dan Roth. 2003. Kernel Methods for Relational Learning. In *Proceedings of ICML 2003*.

Hal Daumé III and Daniel Marcu. 2004. Np bracketing by maximum entropy tagging and SVM reranking. In *Proceedings of EMNLP'04*.

Daniel Gildea and Daniel Jurafsky. 2002. Automatic labeling of semantic roles. *Computational Linguistics*, 28:245–288.

Hans P. Graf, Eric Cosatto, Leon Bottou, Igor Durdanovic, and Vladimir Vapnik. 2004. Parallel support vector machines: The cascade svm. In *Neural Information Processing Systems*.

Isabelle Guyon and André Elisseeff. 2003. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182.

David Haussler. 1999. Convolution kernels on discrete structures. Technical report, Dept. of Computer Science, University of California at Santa Cruz.

T. Joachims. 2000. Estimating the generalization performance of a SVM efficiently. In *Proceedings of ICML'00*.

Hisashi Kashima and Teruo Koyanagi. 2002. Kernels for semi-structured data. In *Proceedings of ICML'02*.

Jun'ichi Kazama and Kentaro Torisawa. 2005. Speeding up training with tree kernels for node relation labeling. In *Proceedings of HLT-EMNLP'05*.

Taku Kudo and Yuji Matsumoto. 2003. Fast methods for kernel-based text analysis. In *Proceedings of ACL'03*.

Taku Kudo, Jun Suzuki, and Hideki Isozaki. 2005. Boosting-based parse reranking with subtree features. In *Proceedings of ACL'05*.

Alessandro Moschitti and Cosmin Bejan. 2004. A semantic kernel for predicate argument classification. In *CoNLL-2004*, Boston, MA, USA.

Alessandro Moschitti, Daniele Pighin, and Roberto Basili. 2006. Semantic role labeling via tree kernel joint inference. In *Proceedings of CoNLL-X*, New York City.

Alessandro Moschitti, Daniele Pighin, and Roberto Basili. 2008. Tree kernels for semantic role labeling. *Computational Linguistics*, 34(2):193–224.

Alessandro Moschitti. 2006. Making tree kernels practical for natural language learning. In *Proccedings of EACL'06*.

Julia Neumann, Christoph Schnorr, and Gabriele Steidl. 2005. Combined SVM-Based Feature Selection and Classification. *Machine Learning*, 61(1-3):129–150.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Comput. Linguist.*, 31(1):71–106.

J. Pei, J. Han, Mortazavi B. Asl, H. Pinto, Q. Chen, U. Dayal, and M. C. Hsu. 2001. PrefixSpan Mining Sequential Patterns Efficiently by Prefix Projected Pattern Growth. In *Proceedings of ICDE'01*.

Alain Rakotomamonjy. 2003. Variable selection using SVM based criteria. *Journal of Machine Learning Research*, 3:1357–1370.

Libin Shen, Anoop Sarkar, and Aravind k. Joshi. 2003. Using LTAG Based Features in Parse Reranking. In *Proceedings of EMNLP'06*.

Jun Suzuki and Hideki Isozaki. 2005. Sequence and Tree Kernels with Statistical Feature Mining. In *Proceedings of the 19th Annual Conference on Neural Information Processing Systems (NIPS'05)*.

Ivan Titov and James Henderson. 2006. Porting statistical parsers with data-defined kernels. In *Proceedings of CoNLL-X*.

Kristina Toutanova, Penka Markova, and Christopher Manning. 2004. The Leaf Path Projection View of Parse Trees: Exploring String Kernels for HPSG Parse Selection. In *Proceedings of EMNLP 2004*.

Vladimir N. Vapnik. 1998. *Statistical Learning Theory*. Wiley-Interscience.

Jason Weston, Sayan Mukherjee, Olivier Chapelle, Massimiliano Pontil, Tomaso Poggio, and Vladimir Vapnik. 2001. Feature Selection for SVMs. In *Proceedings of NIPS'01*.

Jason Weston, André Elisseeff, Bernhard Schölkopf, and Mike Tipping. 2003. Use of the zero norm with linear models and kernel methods. *J. Mach. Learn. Res.*, 3:1439–1461.

Mohammed J Zaki. 2002. Efficiently mining frequent trees in a forest. In *Proceedings of KDD'02*.