A Memory Efficient Graph Kernel

Giovanni Da San Martino, Nicolò Navarin and Alessandro Sperduti
Department of Mathematics
University of Padova, Padova, Italy
Email: {dasan,nnavarin,sperduti}@math.unipd.it

Abstract—In this paper, we show how learning models generated by a recently introduced state-of-the-art kernel for graphs can be optimized from the point of view of memory occupancy. After a brief description of the kernel, we introduce a novel representation of the explicit feature space of the kernel based on an hash function which allows to reduce the amount of memory needed both during the training phase and to represent the final learned model. Subsequently, we study the application of a feature selection strategy based on the F-score to further reduce the number of features in the final model. On two representative datasets involving binary classification of chemical graphs, we show that it is actually possible to sensibly reduce memory occupancy (up to one order of magnitude) for the final model with a moderate loss in classification performance.

I. INTRODUCTION

Kernel methods are very effective tools for many Pattern Recognition and Machine Learning tasks, such as classification and regression. In addition to guarantee state-of-the-art performances they turn also to be very flexible since, given an input domain, it is sufficient to define a proper kernel function on couples of inputs to allow their application. Because of that, it is possible to deal not only with vectorial input spaces, but also with structured domains such as tree or graph domains. For example, chemical compounds can be naturally described via their molecular graphs, where vertices represent atoms and edges represent chemical bonds. Moreover, both discrete and continuous information can be associated to each vertex, e.g. atom type (carbon, oxygen, nitrogen, ...) is an example of discrete information, while physical and chemical properties of an atom type (such as atomic mass average, conductivity, ...) are examples of numerical information. Additional information can be attached to the edges, e.g. type of chemical bond (covalent, ionic, ...). Thus the input domain for chemical compounds can be considered the combination of several discrete and vectorial spaces in a non trivial way. This makes the definition itself of a kernel function for structured domains a complex task. Specifically, while a kernel function defined for a vectorial space typically involves a trivial amount of computation to compute the kernel value, kernels for structured domains have to face non trivial computational challenges. For example, if we restrict our discussion to kernel for graphs\(^1\), a nowadays well understood issue is the trade-off between kernel expressivity versus efficiency [1]: a kernel able to capture all the small structural differences between two graphs is doomed to be inefficient from a computational point of view, i.e. an efficient algorithm\(^2\) for computing it is not known. So, if we want to define a kernel for graphs that is efficient to compute, we have to give up on the possibility to discriminate among all different graphs. Because of that, the definition of a kernel for graphs that is both efficient to compute and expressive enough to give satisfactory performances is a very challenging task. The main adopted strategy consists in decomposing a graph into a collection of simpler substructures \(s_i(g)\) for which it is easier to define an efficient kernel \(k_s(\cdot,\cdot)\) and then define the kernel between two graphs \(k(g_1, g_2)\) as the combination of the result of all the kernel computations on all the couples of substructures belonging to the two graphs, i.e. \(k_s(s_i(g_1), s_j(g_2)) \forall i,j\). Examples of such substructures are walks [1], paths [2], [3], or subtrees [4], [5]. Another approach consists in restricting the input domain to a class of graphs for which it is possible to efficiently perform a check on differences (see, for example, [6].) Some of the above mentioned kernels are quite efficient and perform well on specific domains. However, their adoption in an industrial settings, where a large quantity of data needs to be processed, is still problematic. In fact, the use of these kernels in conjunction with Kernel Methods, such as Support Vector Machines [7], typically produces models involving a significant number of support graphs due to the complexity of the classification (e.g., "is a chemical compound toxic or not?") or regression (e.g., "what is the quantity of a drug that needs to be used to obtain a reference biological response?") tasks, leading to the occupancy of a non trivial amount of memory and consequent increase in the time of computation of the model output for graphs to be evaluated. One possible solution to this problem is to try to work on an explicit feature space, so to avoid the storage of the support graphs after training. In order to have a more compact model, however, the explicit feature space needs to be very expressive and small, otherwise it may turn more efficient and effective to keep the support graphs. Two possible ways to define an expressive and efficient feature space for structures have been explored in the past. The first approach is based on the adoption of graph mining techniques for discovering the \(k\)-top most frequent graphs in the dataset. Structural features are then extracted from these

\(^1\)Graphs constitute a very general class of structures, allowing the representation of very many structured domains of practical interest. It should be noticed that sequences and trees are special cases of graphs, so structured domains emerging from Natural Language Processing as well as Bioinformatics are fully covered by focusing on graphs.

\(^2\)An algorithm is considered efficient if it can compute the required output in time that is polynomial with respect to the input dimension.
graphs and used as explicit feature space to be exploited by any standard machine learning approach, including Kernel Methods. An example of this approach is gBoost [8], where the explicit feature space is used by a boosting algorithm. One drawback of this approach is that the selection of the explicit structural features is done via a mining technique disregarding the classification/regression task at hand, so it is quite possible that significant structural features are non included in the selected ones. An alternative approach consists in exploiting the fact that structural features typically share components, e.g. if a graph is represented by trees which can be obtained by breadth-first visits starting from each vertex of the graph, it is very likely that a given subtree occurs in many of these trees. Hinging on this observation, it may be possible to represent explicitly a very expressive feature space in a very compact way, simultaneously allowing fast computation of kernel values [9] (ODD kernels). The advantage of this approach consists in the fact that all the structural features represented in the training set are retained, thus avoiding the risk to discard potentially useful features before performing learning. In [9], however, it is only shown that the proposed kernel is both effective and fast to compute. No proposal on how to reduce the demand of storage resources is done.

In this paper, we start from the kernel defined in [9], and show how the explicit representation of the feature space can actually be exploited to perform feature selection by considering the specific task at hand. Specifically, on datasets involving chemical compounds, we study the adoption of the F-score to select the most significant features and we show that it is actually possible to significantly reduce the size of the feature space by preserving quite good performances. We are also able to explicitly show, for the studied datasets, the cumulative distribution of the F-score values with respect to the structural features as well as the impact of feature selection on the number of support graphs involved in the final model. To our best knowledge this is the first time that this kind of studies are performed on domains of graphs. Finally, we discuss how to efficiently compact the model obtained after feature selection and training, so to lead to a reduced memory usage while maintaining an efficient computation of the output score for graphs to be evaluated.

II. AN ODD-KERNEL FOR GRAPHS

This section describes an ODD-Kernel for graphs. Let us first introduce some notation. A graph $G(V,E,L)$ is a triplet where $V$ is the set of vertices, $E$ the set of edges and $L()$ a function mapping nodes to labels. A path $p(v_i,v_j)$ of length $n$ in a graph $G$ is a sequence of nodes $v_1,\ldots,v_n$, where $v_1 = v_i$, $v_n = v_j$ and $(v_k,v_{k+1}) \in E$ for $1 \leq k < n$. A tree is a directed acyclic graph where each node has at most one incoming edge. The root of a tree $T$ is represented by $r(T)$. A proper subtree rooted at node $v$ comprises $v$ and all its descendants.

In order to describe the kernel, the framework on which it is based on is first introduced. The strategy for defining kernels that the framework employs is to decompose the input graph into a set of substructures for which the test of isomorphism can be performed efficiently, i.e. subtrees. This is ultimately obtained by mapping the input graphs into two sets of trees and then applying a kernel for trees on each pair of trees in the two sets. In order to map the graphs into trees, two intermediate steps are needed (see down in the section for further details). In the first one, the graph is decomposed into a multiset of unordered DAGs. The decomposition ensures that isomorphic graphs are represented by the same multisets of DAGs [9]. Since the vast majority of tree kernels, including the one we are going to use in the paper, are defined for ordered trees, an ordering between DAG nodes is defined. Finally, any Ordered DAG (ODD) is mapped into a multiset of trees by performing a tree visit of the DAG (see Fig. 2 for an example). Among the kernels for trees defined in literature, the one employed in the paper is the Subtree Kernel [10], which counts the number of shared proper subtrees between the two input trees. The Subtree Kernel is defined as follows:

$$K(T_1, T_2) = \sum_{i=1}^{s} \lambda^{|i|} \phi_i(T_1) \phi_i(T_2) = \sum_{v_1 \in T_1} \sum_{v_2 \in T_2} C(v_1, v_2),$$

where $\phi_i(T)$ is the frequency of the i-th feature, i.e. the i-th proper subtree (whose number of nodes is $|i|$) in $T$, $\lambda$ is a parameter downweighting the influence of larger subtrees and $C()$ is a function counting the number of shared proper subtrees rooted at $v_1$ and $v_2$. $C(t_1,t_2)$ can be recursively computed according to these rules:

1) if $t_1$ and $t_2$ are different then $C(t_1,t_2) = 0$.
2) if $t_1 = t_2$ and $t_1$ is a preterminal then $C(t_1,t_2) = \lambda$.
3) if $t_1 = t_2$ and $t_1$ is not a preterminal then $C(t_1,t_2) = \lambda \prod_{j=1}^{nc(t_1)} C(ch_j[t_1],ch_j[t_2])$, where $nc(t_1)$ is the number of children of $t_1$ and $ch_j[t]$ is the $t$-th child of node $t$.

The computational complexity of eq. (1) is $|T| \log |T|$.

The following scheme summarizes the steps for defining the kernel for graphs (see [9] for further details):

1) map $G$ into a multiset of DAGs $\{DD_G^{v_i} | v_i \in V_G\}$, where $DD_G^{v_i}$ is obtained by keeping each edge in the shortest path(s) connecting $v_i$ with any $v_j \in V$. Fig. 1 gives an example of DAG decomposition of a graph. It is possible to show that isomorphic graphs are represented exactly by the same multiset of DAGs $DD_G$. Since the visit is repeated for every node in $G$, any edge, which is not
reduce storage occupancy. In addition, we also discuss how representing the feature space with respect to the solution

In this section, we propose a related but novel way of ordering the nodes to be ordered and $DD^v_G$ ones are not, a strict partial order between DAG nodes has been defined in [9] yielding Ordered Decompositional Dags $ODD^v_G$. Such ordering ensures that the swapping of non comparable nodes does not change the feature space representation of the examples, i.e. the map into feature space related to the kernel is well defined. A more efficient ordering will be defined in Section III.

3) Let us define $T(v_i)$ as the tree resulting from the visit of $ODD^v_G$ starting from node $v_i$. Fig. 2 gives an example of tree visits of two DAGs. Notice that any node $v_j$ of the DAG having $l > 1$ incoming edges will be duplicated $l$ times in $T(v_i)$ (together with all the nodes that are reached from $v_j$ in the same visit). Now the set $ODD^v_G$ has been transformed into a multiset of trees and thus, the kernel of eq. (1) can be applied as

$$K_{DAG}(D_1, D_2) = \sum_{v_1 \in V_{D_1}} \sum_{v_2 \in V_{D_2}} C((T(v_1)), (T(v_2))),$$

Notice that the tree visit $T(v_i)$ can be stopped when the tree $T(v_i)$ reaches a maximum depth $h$. Such tree is referred to as $T_h(v_i)$. In order to control the computational complexity and add expressiveness to the kernel, all the contribution for every $T_j(v_i)$, with $1 \leq j \leq h$ can be summed:

$$K_{DAG}(D_1, D_2) = \sum_{v_1 \in V_{D_1}} \sum_{v_2 \in V_{D_2}} C((T_j(v_1)), (T_j(v_2))).$$

Finally, a kernel between graphs which uses the DAG decomposition shown in this section and the Subtree Kernel for trees can be defined as

$$K(G_1, G_2) = \sum_{D_1 \in DD(G_1)} \sum_{v_1 \in V_{D_1}} \sum_{D_2 \in DD(G_2)} \sum_{v_2 \in V_{D_2}} C((T_j(v_1)), (T_j(v_2)))$$

The kernel of eq. (5) is the one which is used in the remainder of the paper.

III. FEATURE SPACE REPRESENTATION VIA HASHMAP

In this section, we propose a related but novel way of representing the feature space with respect to the solution adopted in [9]. The new representation allows to drastically reduce storage occupancy. In addition, we also discuss how to trivally reduce storage occupancy for the model obtained after training of an SVM.

Instead of representing explicitly the structured feature space of the kernel, that may be inefficient in terms of space and time complexity, we can use an implicit representation. Specifically, representing the various trees that form the feature space may not be the best choice when we have to search in the feature space for a specific input feature. Moreover, representing each tree by some data structure that maintains the information about the nodes (and the pointers to that nodes) introduces an additional space overhead.

For our purposes, if we have to compute the kernel value between two graphs, we only need to know how many features the two graphs have in common and the size of the shared features, since this information is used as the exponent of the $\lambda$ parameter in the kernel formula. So we do not need to keep an explicit representation of the structure corresponding to a given feature as long as we are able to code the feature in a unique way\(^3\), e.g. by using a function that returns an identifier (e.g., a numerical value) given the structural representation of the feature in input. If we are able to define such function, then we can compare the features present into two graphs by comparing their identifiers. Let us elaborate on how such a function can be defined. Let consider an hash function $\kappa(\cdot)$ which given an alphanumeric string in input returns an integer value. Let for the moment assume that $\kappa(\cdot)$ is perfect, i.e. no repeating hash values will arise from different alphanumeric strings in input. Then, given a vertex $v$, we can assign the following alphanumeric string $\sigma(v)$ to it:

$$\sigma(v) = L(v) \cdot \left( \prod_{i=1}^{\text{outdeg}(v)} k(\kappa(ch_i[v])) \right)$$

\(^3\)Actually, features that are structurally different, but which cannot be discriminated by the used kernel can get identical code. For example a diamond shape graph and the tree obtained by a breadth-first visit of the same graph, as a whole, constitute equivalent features from the kernel point of view.
where ‘·’ is the concatenation operator\(^4\) and ‘\(\prod\)’ represents the application of ‘·’ to an indexed sequence of arguments; moreover, the order adopted for the children \(ch_i[v]\) of \(v\) is the same order induced by the hash function \(\kappa(\cdot)\), i.e., \(v_1\) comes before \(v_2\) if \(\kappa(\sigma(v_1)) < \kappa(\sigma(v_2))\). Notice that the above function is well defined since if \(v\) is a leaf, then the alphanumeric string associated to it is given by \(L(v)\); moreover, strings associated to leaf nodes are sorted by the \(\kappa(\cdot)\) function, which assigns different numerical values to different labels; in turn, such order can be exploited to apply the \(\sigma(\cdot)\) function to internal nodes that only have leaf nodes as children and recursively to other internal nodes.

It should be noticed that:

- \(\sigma(v)\) can efficiently be generated from \(L(v)\) and the strings generated for its children;
- the size of each string is bounded by a constant which in practice can be considered independent from the number of descendants of \(v\);
- no explicit extended representation of the features is generated.

Thus, any feature can be identified by the value returned by \(\kappa(\cdot)\) applied to the string obtained via \(\sigma(\cdot)\), and stored into an hash table in association with additional information, such as feature size and how frequent that feature occurs into any of the training graphs. Actually, we can construct the whole feature space representation directly from the tree-visits of the graphs (one starting from each node) in the training set. In fact, if we insert in the table all the nodes belonging to each tree, starting from the leaves up to the roots, we are sure that all the graph features are inserted into the table in an efficient and well defined way. Statistics concerning feature size and frequency for each graph can be incrementally and efficiently collected as well. Specifically, for each different alphanumeric string \(\sigma(v)\) (where \(v\) is a node belonging to one of the training graph \(g_i\)) the corresponding entry in such hash table is the following tuple:

\[
(\text{key} = \tau(\sigma(v)), \text{size}_\text{visit}(\sigma(v)), \#(\sigma(v), g_1), \ldots, \#(\sigma(v), g_n))
\]

where, \(\text{size}_\text{visit}(\sigma(v))\) is equal to the number of nodes which have been visited to generate \(\sigma(v)\), and \(\#(\sigma(v), g_i)\) is equal to the number of times \(\sigma(v)\) has been generated when visiting nodes in the training graph \(g_i\). It can be observed that

\[
\text{size}_\text{visit}(\sigma(v)) = 1 + \sum_{i=1}^{\text{outdeg}(v)} \text{size}_\text{visit}(\sigma(ch_i[v])),
\]

and that the hash table will contain all the information concerning the training set: the representation of a graph \(g_i\) in the feature space is obtained by projecting all the touples on \(\#(\sigma(v), g_i)\). Moreover, the kernel value between graphs \(g_i\) and \(g_j\) can be computed as

\[
K(g_i, g_j) = \sum_{f \in \text{hash-table}} \lambda^{\text{size}_\text{visit}(f) \#(f, g_i) \#(f, g_j)}.
\]

\(^4\)Here we assume that the concatenation operator inserts between the two concatenated strings a special character non belonging to the alphabet used to generate the strings, so to avoid to generate the same string starting from different input strings, e.g. concatenation of label "a1" with hash value "1" would otherwise lead to the same string obtained by concatenating label "a" with hash value "11".

Finally, after training with an SVM, the optimal weight vector can be obtained by collapsing all the contributions of the support graphs for each feature \(f\), i.e.,

\[
w_f^* = \lambda^{\frac{\text{size}_\text{visit}(f)}{2}} \sum_{i \in SG} y_i \alpha_i^* \#(f, g_i),
\]

where \(SG\) is the set of support graphs, \(y_i\) is the target value for \(g_i\), and \(\alpha_i^*\) is the optimal value for the dual variable associated to \(g_i\).

Thus, after training, the model can be represented by an hash table with the following entries:

\[
(\text{key} = \tau(f), w_f^*)
\]

which drastically reduces the storage occupancy. Notice that, given a graph \(\tilde{g}\) for which the score with respect to the model has to be computed, the contribution of each feature occurring in \(\tilde{g}\) can be computed by generating the corresponding alphanumeric string \(f\) via \(\sigma(\cdot)\), and then looking for \(\tau(f)\) into the model hash table for checking if there is a match. In case of match, the corresponding \(w_f^*\) is used to update the score\(^5\). Thus the score computation can be performed in time and space which is linear with respect to the number of nodes generated by the tree-visits of \(\tilde{g}\).

The above approach leads to practical computational improvements with respect to an approach where the features are represented explicitly since the feature space is constructed directly from the graphs with no need to define an order between nodes being such order induced by the hash function \(\kappa(\cdot)\).

As a final remark on the requirement for \(\kappa(\cdot)\) to be a perfect hash function, it must be considered that in real-world datasets, the number of features is relatively small compared to the space of different (non perfect) hash codes, so the probability that two features are assigned to the same hash value is very small and depends on the complexity of the selected hash function. In addition to that, if an SVM/SVR is used, only the features occurring in the support graphs must be considered, further reducing the probability of a collision for the hash function.

### IV. Feature Selection

One nice characteristic of the described kernel is the possibility to perform feature selection thanks to the explicit representation of the feature space. Feature selection consists in deleting non-informative features in order to reduce noise and to increase performance. In our case, feature selection is appealing since it can reduce the number of features to store in memory for representing the model generated via learning. Of course, a significant reduction in number of features should not reduce too much the performance of the

\(^5\)If score is the partial value of the score computed till the matching feature \(f\) is considered, the updated value for the score becomes

\[\text{score} \leftarrow \text{score} + \lambda^{\frac{\text{size}_\text{visit}(f)}{2}} w_f^*\].

In order to avoid to recompute \(\text{size}_\text{visit}(f)\) for features in \(\tilde{g}\), it is more convenient to store into the model hash table the weight \(\tilde{w}_f^* = \lambda^{\frac{\text{size}_\text{visit}(f)}{2}} w_f^*\) and update the current score according to

\[\text{score} \leftarrow \text{score} + \tilde{w}_f^*\].
model, which otherwise becomes useless. The application of feature selection in the context of kernels for graphs is not usual due to the implicit definition of the feature space. Only when adopting other learning methods for graphs, feature selection has been applied (e.g. [8]) but just on the basis of frequency occurrence and disregarding target information.

In the literature, several feature selection principles have been proposed (see [12] for a description of many of them). A typical approach is to compute a statistical measure for estimating the relevance of each feature w.r.t. the target concept, and to discard the less-correlated features.

In order to do as well as possible, the measure to apply should take advantage of all available information, that in our learning scenario means to care about both feature frequency and target information. We have decided to study one measure that possesses these characteristics, the F-score. This is a measure that has been widely used in various tasks including, for example, information retrieval.

The F-score [12] of a feature \( i \) is defined for binary classification tasks as follows:

\[
F_s(i) = \frac{(AVG_i^+ - AVG_i^+)^2 + (AVG_i^+ - AVG_i^-)^2}{\sum_{j \in Tr^+} (N_j^+ - AVG_i^+)^2 + \sum_{j \in Tr^-} (N_j^+ - AVG_i^-)^2}
\]

where \( AVG_i^+ \) is the average frequency of feature \( i \) in the dataset, \( AVG_i^+ - AVG_i^- \) is the average frequency of feature \( i \) in positive (negative) examples, \( |Tr^+|/|Tr^-| \) is the number of positive (negative) examples and \( N_j^+ \) is the frequency of feature \( i \) in the \( j^{th} \) example of the dataset. It should be noticed that features that get small values of F-score are not very informative with respect to the binary classification task. Thus, features with F-score below a given threshold can be removed. Given a predefined tolerance on the drop of classification performance, a suitable value for such threshold can be estimated using a validation set or a cross-validation approach.

V. EXPERIMENTS

This section analyzes, from a practical point of view, the effects of reducing the size of the model by making use of its explicit representation and by applying the feature selection technique based on the F-score (eq. (6)). Two chemical compound datasets were used in the experiments: CAS\(^6\) and AIDS [13]. Both datasets represent binary classification problems. CAS is a mutagenicity dataset. It comprises 4,337 graphs, where the average number of nodes per graph is 29.9 and the average number of edges is 30.9. The percentage of positive examples is 55.36%. AIDS is an antiviral screen dataset. It comprises 1,503 graphs. The average number of nodes per graph is 58.9 and the average number of edges is 61.4. The percentage of positive examples is 28.07%.

All the experiments in this section involve the Support Vector Machines together with the kernel defined in eq. (5).

We recall that on these datasets this kernel (with \( h = 4 \)) obtains state-of-the-art classification performances. The feature selection process is performed by choosing a set of threshold values and then discarding all those features whose F-score, computed according to eq. (6), is lower than such threshold.

We first analyzed the size of the model, in terms of number of features, after the application of various threshold values. Fig. 3 and Fig. 4 report the resulting plots. The number of discarded features is not linear with respect to the threshold value, in particular there are high non-linearities around the values 0.00023 and 0.0003 for CAS and one around the value 0.0006 for AIDS.

We then checked how the accuracy of the learning algorithm was affected when reducing the number of features through the F-score. We performed a 10-fold cross validation on the datasets and reported mean accuracies and corresponding standard deviations. Fig. 5 plots the values of accuracy and

\(^{6}\)http://www.cheminformatics.org/datasets/bursi
standard deviation with respect to the F-score threshold and the number of features which survived the selection process. Notice that between threshold values 0 and 0.0003 the accuracy decreases from 0.847 to 0.839, while the number of features retained decrease from 40,495 to 11,549 on average. This means that the size of the model, i.e. the number of features, is reduced to about \( \frac{1}{4} \), with no significant loss in accuracy. Increasing the threshold to 0.001, the associated accuracy becomes 0.832 and the number of features becomes 3,477, i.e. less than \( \frac{1}{17} \) of the original size with a total accuracy loss of 0.015. After the threshold value 0.005 the number of features is 441, but the accuracy drops to 0.789. Clearly, too many significant features are discarded and the learning algorithm is not able to create a meaningful model. A similar behaviour can be observed in Fig. 6 for the AIDS dataset. However, notice that the 0.0003 threshold has a slight accuracy improvement with respect to 0 (from 0.825 to 0.829) and the number of features decreases from 30,843 to 27,169.

Fig. 6 does not show the behaviour for higher F-score values, however it is interesting to notice that with threshold value set at 0.05 the accuracy is 0.785, so 0.042 less than the reference figure, but with just 90 features.

Fig. 7 plots the number of support graphs as a function of the F-score thresholds. Notice that, for threshold values less than 0.001, the number of support graphs drops, while there is no significant accuracy loss. This means that the learning algorithm is able to produce a simpler but effective model.

One can argue that eliminating features can lead a graph to lose all of its features in the representation. This graph then will not be represented at all in the model. Fig. 8 shows that this happens only for high F-score values, so this does not happen with the range of F-score values we considered acceptable looking at the accuracy.

A final remark that is worth to make is the following. If the number of explicit features needed to represent a model is larger than the total number of vertices in the support graphs, then it is more convenient from the storage point of view to use the representation of the model in the dual space. We have verified that this is not the case for the two studied datasets. In fact, for the CAS datasets, the total number of vertices in the support graphs without feature selection is around 240,300, while for the AIDS dataset, the same number is around 146,158. Of course, these values are far larger than the number of features retained by the models, with or without feature selection.

**VI. CONCLUSIONS**

Graphs can be very high dimensional data structures. Since the model produced in the learning phase by kernel methods may consist of a significant number of support vectors, the required memory can prevent the application of such learning techniques to real-world problems. When the kernel involved can be represented explicitly as a list of features, such as the one described in this paper, the memory requirements...
can be greatly reduced and feature selection techniques effectively applied. The paper discusses the application of the F-score feature selection technique and its impact on memory requirements, speed and accuracy showing that an effective reduction of memory allocation can be achieved with no significant speed and accuracy loss. Future works will include the application of the proposed technique to other kernels that allow for an explicit representation of the feature space, and the study of additional feature selection techniques. A further research direction involves the assessment of the significance, from the biological point of view, of the selected features.

VII. ACKNOWLEDGMENTS

This work was supported by the Italian Ministry of Education, University, and Research (MIUR) under Project PRIN 2009 2009LNP494_005.

REFERENCES


