Tree-Based Kernel for Graphs With Continuous Attributes

Giovanni Da San Martino, Nicolò Navarin, *Member, IEEE*, and Alessandro Sperduti, *Senior Member, IEEE*

***Abstract*— The availability of graph data with node attributes that can be either discrete or real-valued is constantly increasing. While existing Kernel methods are effective techniques for dealing with graphs having discrete node labels, their adaptation to nondiscrete or continuous node attributes has been limited, mainly for computational issues. Recently, a few kernels especially tailored for this domain, and that trade predictive performance for computational efficiency, have been proposed. In this brief, we propose a graph kernel for complex and continuous nodes' attributes, whose features are tree structures extracted from specific graph visits. The kernel manages to keep the same complexity of the state-of-the-art kernels while implicitly using a larger feature space. We further present an approximated variant of the kernel, which reduces its complexity significantly. Experimental results obtained on six real-world data sets show that the kernel is the best performing one on most of them. Moreover, in most cases, the approximated version reaches comparable performances to the current state-of-the-art kernels in terms of classification accuracy while greatly shortening the running times.**

***Index Terms*— Big data applications, machine learning, supervised learning, support vector machines.**

I. Introduction

There is an increasing availability of data in the form of attributed graphs, i.e., graphs where some information is attached to nodes and edges (and to the graph itself). For computational reasons, the available machine learning techniques for graph-structured data have been focusing on problems whose data can be modeled as graphs with discrete attributes. However, in many application domains, such as bioinformatics and action recognition, nondiscrete node attributes are available [1], [2]. For example, many bioinformatics problems deal with proteins. It is possible to represent a protein as a graph, where nodes represent secondary structure elements. Two nodes are connected whenever they are neighbors either in the amino acid sequence or in space [1]. Each node has a discrete-valued attribute, indicating the structure it belongs to (helix, sheet, or turn). Moreover, several chemical and physical measurements can be associated with each node, such as the length of the secondary structure element in Å, its hydrophobicity, polarity, polarizability, and so on. In some tasks, discarding such type of information has a significantly negative impact on the predictive performance (see Section VI).

Most of the graph kernels in the literature are not suited for nondiscrete node labels, since their computational efficiency hinges on hard matches between discrete labels. Of course, this strategy cannot work for nondiscrete labels, which in general are distinct. An alternative would be to define a kernel function between graph nodes; however, the resulting computational times become unfeasible, as in the case of [3]. For this reason, recently there has been an increasing interest in the definition of graph kernels that can efficiently deal with continuous-valued attributed graphs. The problem is challenging because both fast and expressive kernels (in terms of discriminative power) are looked for.

In this brief, we present a new kernel inspired by the graph kernel framework proposed in [4]. The features induced by the kernel are tree structures extracted from breadth-first visits of a graph (contrary to [1] an edge is only traversed once per visit). We extend the definition of tree kernels, and consequently derive a graph kernel, which is able to deal with complex and continuous node labels. As the experimental results show, the richer feature space allows reaching the state-of-the-art classification performances on real-world data sets. While the computational complexity of our kernel is the same as competing ones in the literature, we describe an approximated computation of the kernel between graph nodes in order to reach lower running times, while keeping the state-of-the-art results.

Manuscript received May 18, 2016; revised December 20, 2016; accepted May 10, 2017. Date of publication June 13, 2017; date of current version June 21, 2018. This work was supported by the University of Padua under the strategic project BIOINFOGEN. *(Corresponding author: Nicolò Navarin.)*

G. Da San Martino is with the ALT Research Group, Qatar Computing Research Institute, Hamad Bin Khalifa University, Doha 5825, Qatar.

N. Navarin and A. Sperduti are with the Department of Mathematics, University of Padua, 35121 Padova, Italy (e-mail: nnavarin@math.unipd.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNNLS.2017.2705694

II. Notation

A graph $G = (V_G, E_G, and L_G)$ is a triplet, where V_G is the set of n vertices, E_G the set of m edges, and $L_G()$ a function mapping nodes to discrete labels. A graph is undirected if $(v_i, v_j) \in E_G \Leftrightarrow (v_j, v_i) \in E_G$, otherwise it is directed. A path $p(v_i, v_j)$ of length s in a graph G is a sequence of nodes $u_0, \ldots, u_{s-1}$, where $u_i \in V_G$, $u_0 = v_i$, $u_s = v_j$, and $(u_k, u_{k+1}) \in E_G$ for $0 \leq k < s-1$. A cycle is a path for which $u_0 = u_{s-1}$. A graph is acyclic if it has no cycles. A tree is a directed acyclic graph where each node has exactly one incoming edge, except the root node which has no incoming edge. The root of a tree T is represented by $r(T)$. The ith child (outgoing edge) of a node $v \in V_T$ is referred to as $ch_v[i]$. The number of children of a node v is referred to as $\rho(v)$ (ρ is the maximum out-degree of a tree or graph). A leaf is a node with no children. A proper subtree rooted at node v comprises v and all its descendants.

III. Related Work

In the last few years, several graph kernels for discrete-valued graphs have been proposed in the literature. Early works presented kernels that have to be computed in closed form, such as the random walk kernel [5] or the shortest path kernel [3]. These kernels suffer from a relatively high computational complexity: $O(n^3)$ and $O(n^4)$, respectively. More recently, research focused on the efficiency of kernel calculation. The state-of-the-art kernels use explicit feature mapping techniques [4], [6], [7], with computational complexities almost linear in the size of the graphs. If we consider graphs with continuous-valued labels, this last class of kernels cannot be easily modified to deal with them, because their efficiency hinges on the ability to perform computation only for discrete labels that match. Of course, this is not possible when considering continuos-valued labels. Between the two obvious possible solutions, i.e., adopt slower kernels or discretizing/ignoring the continuous attributes of the graphs, the latter approach was usually the preferred one [8]. In [9], a kernel for graphs with continuous-valued labels has been presented. The kernel matches common subgraphs up to a fixed size k, and has complexity $O(n^k)$.

In [10], another more efficient kernel has been presented. This kernel is a sum of path kernels that in turn are a sum of node

2162-237X © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: POLO BIBLIOTECARIO DI INGEGNERIA. Downloaded on June 03,2020 at 07:28:49 UTC from IEEE Xplore. Restrictions apply.

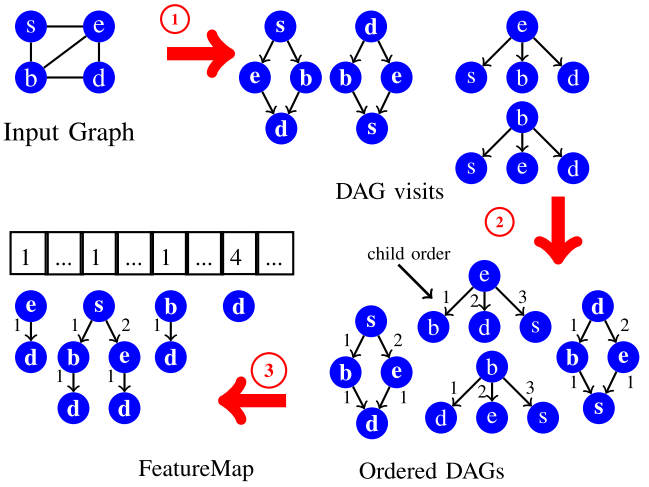

Fig. 1. ODD kernel summary. 1: decomposition of a graph into its DDs. 2: definition of a total ordering among the children of each node. 3: generation of an explicit FeatureMap extracting all proper subtrees (ST kernel) from the set of ordered DDs.

kernels. The computational complexity of the kernel is $O(n^2(m + \log n + d + \sigma^2))$, where n and m are the number of nodes and edges in the graph, respectively, σ is the depth of the graph, and d is the dimension of the vectors associated with nodes. However, experimental results show that this kernel cannot achieve the same predictive performance as other computationally more demanding graph kernels, e.g., the shortest path kernel.

Very recently, two kernel frameworks able to deal with continuous and vectorial labels have been proposed: Neumann *et al.* [11] propose to use locality sensitive hashing to discretize continuous and vectorial labels, while in [12], a very general framework of graph kernels is proposed.

The experience on discrete-labeled graphs teaches us that path features are not the most expressive ones. In fact, in [4], [7], [13], and [14], it is shown that tree features can express a more suitable similarity measure for many tasks. The framework presented in [4] is especially interesting, since it allows to easily define a kernel for graphs from a vast class of tree kernels, and it constitutes the starting point of our proposal.

IV. Ordered Decomposition DAG Kernels for Graphs With Discrete Labels

The kernel we are going to propose is based on tree structures. This section briefly recalls the procedure for extracting them from a graph [4], [15]. In order to map the graphs into trees, two intermediate steps are needed.

1) Map the graph G into a multiset of decomposition Directed Acyclic Graph (DAG)s $\mathrm{DD}_G = \{\mathrm{DD}_G^{v_i} | v_i \in V_G\}$, where $\mathrm{DD}_G^{v_i}$ is formed by the nodes and the directed edges in the shortest path(s) between v_i and any $v_j \in V_G$. Fig. 1(1) shows an example of DD_G. In order to reduce the time required for evaluating the kernel, the visits can be restricted to those nodes whose shortest path length with respect to v_i is not greater than a parameter h. We recall the following facts discussed in more detail in [4]. Let $H_{\max}$ be the maximum number of nodes of a $\mathrm{DD}_G^{v_i} \in \mathrm{DD}_G$, then $H_{\max} \leq \lfloor ((\rho^{h+1} + 1)/(\rho - 1)) \rfloor \leq n$. The decomposition we have defined ensures that isomorphic graphs are represented by the same multiset of DAGs, which is a necessary condition for the kernels, we will propose to be well defined. The computation of the multiset DD_G for a graph G requires $O(nm)$ time.
2) Since the kernel we are going to describe in this brief requires the DAG nodes to be ordered, a strict partial order between nodes in $\mathrm{DD}_G^{v_i}$ has been defined yielding an ODD $\mathrm{ODD}_G^{v_i}$. The ordering relies on an encoding of the proper sub-DAGs rooted at each node as strings. Let $\kappa : \sum^* \to \sum^w$ and $\pi : V \to \sum^w$ be two functions returning strings of length w. We assume κ to be a perfect hash function for a sufficiently large w. Let $\|$ be the concatenation operator between strings. We then define the encoding function for a node $\pi(\cdot)$ as

$$\pi(v) = \kappa(\kappa(L(v)) \| \pi(ch_v[1]) \| \ldots \| \pi(ch_v[\rho(v)])). \quad (1)$$

When a node v is a leaf, (1) reduces to $\pi(v) = \kappa(L(v))$. The fact that the output of π is of fixed size, and that it is a combination of values returned by a perfect hash function, ensures that it is well defined. A strict partial order between nodes is then the alphanumeric ordering between strings $\pi(\cdot)$. Although there exist different DAGs represented by the same $\pi(\cdot)$ values, the swapping of nodes with the same $\pi(\cdot)$ value does not change the feature space representation of the examples [4]. Fig. 1(2) shows an example of such ordering. Once $\kappa(\cdot)$ values are computed, ordering the sibling of each node in a DD requires $\sum_{v \in \mathrm{DD}} \rho(v) \log \rho(v) \leq \log \rho \sum_{v \in \mathrm{DD}} \rho(v) = m \log \rho$ steps. Ordering all n DAGs in a DD_G requires $O(nm \log \rho)$ time.
3) Finally, any ordered DAG (ODD) is mapped into a multiset of trees. Let us define $T(v_i)$ as the tree resulting from the visit of $\mathrm{ODD}_G^{v_i}$ starting from node v_i: the visit returns the nodes reachable from v_i in $\mathrm{ODD}_G^{v_i}$. If a node v_j can be reached more than once, more occurrences of v_j will appear in $T(v_i)$. In the following, the notation $T_l(v_i)$ indicates a tree visit of depth l. 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). Thus, given a DAG $\mathrm{ODD}_{G_1}^{v_i}$, the total number of nodes of all the tree visits, i.e., $\sum_{v \in \mathrm{ODD}_{G_1}^{v_i}} |T(v)|$, can be exponential with respect to $|V_{G_1}|$. However, such observation does not imply that the complexity of the resulting kernel is exponential since tree visits need not to be explicitly computed to evaluate the kernel function.

The ODD kernel, we are going to use in this brief, is defined as

$$K(G_1, G_2) = \sum_{\substack{OD_1 \in \mathrm{ODD}_{G_1} \\ OD_2 \in \mathrm{ODD}_{G_2}}} \sum_{\substack{v_1 \in V_{OD_1} \\ v_2 \in V_{OD_2}}} \sum_{j=1}^{h} C(r(T_j(v_1)), r(T_j(v_2))) \quad (2)$$

where $C()$ is a recursive function used for computing the subtree kernel (ST) [16], [17]

$$C_{\mathrm{ST}}(v_1, v_2) = \begin{cases} \lambda \quad \text{if } L(v_1) = L(v_2) \wedge v_1, v_2 \text{ are leaves} \\ \lambda \prod_{j=1}^{\rho(v_1)} C_{\mathrm{ST}}(ch_{v_1}[j], ch_{v_2}[j]) \\ \quad \text{if } L(v_1) = L(v_2) \wedge \rho(v_1) = \rho(v_2) \\ 0 \quad \text{otherwise.} \end{cases} \quad (3)$$

Here, λ is a kernel parameter. The ST kernel counts the number of matching proper subtrees between the two input trees. While we focus on the ST kernel in this brief, similar extensions can be easily applied to other tree kernels.

Authorized licensed use limited to: POLO BIBLIOTECARIO DI INGEGNERIA. Downloaded on June 03,2020 at 07:28:49 UTC from IEEE Xplore. Restrictions apply.

V. Graph Kernels for Continuous Node Labels

This section extends the ST kernel to deal with nondiscrete node labels. However, the kernel we describe is also able to deal with continuous labels only. Let us now extend a few definitions to the continuous domain. In order to simplify the presentation, we will also cast the notation and the following function definitions to the domain of the tasks we address in the experimental section. Let us define a graph with continuous attributes as $G = (V_G, E_G, L_G, A_G)$, where $A_G()$ is a function associating to each node a real-valued vector in $\mathbb{R}^d$. In the following, we will assume the DAGs to be ordered as described in Section IV. Let us assume a kernel on continuous attributes $K_A(v_1, v_2)$ is given as parameter. We start in Section V-A by describing a straightforward extension of the kernel for discrete labels presented in Section IV. We then propose an alternative kernel definition in Section V-B. Moreover, in Section V-B, we provide an efficient algorithm for computating the kernel.

A. First Kernel for Graphs With Continuous Node Labels

A straightforward way to extend the ST kernel to deal with continuous labels is to introduce $K_A()$ kernel on continuous labels wherever the two discrete labels match

$$C'_{\mathrm{ST}}(v_1, v_2) = \begin{cases} \lambda \cdot K_A(v_1, v_2) & \text{if } L(v_1) = L(v_2) \wedge v_1, v_2 \text{ are leaves} \\ \lambda \cdot K_A(v_1, v_2) \cdot \prod_{j=1}^{\rho(v_1)} C'_{\mathrm{ST}}(ch_{v_1}[j], ch_{v_2}[j]) & \\ & \text{if } L(v_1) = L(v_2) \wedge \ \rho(v_1) = \rho(v_2) \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

A drawback of this approach is that the kernel value between v_1 and v_2 may be influenced by the function used for ordering the nodes.

For example, assume that v_1 and v_2 have the same number of children, each one with the same discrete label, but with a different continuous label. In this case, the pairs for which K_A is computed, and consequently the value of the kernel evaluation, depends on how identical discrete labels are ordered. Even if we extend the ordering function to consider continuous labels, the selection of pairs would be biased by the ordering function. Ideally, we would like to compute the kernel for all those nodes whose discrete labels are identical. However, extending (4) with such goal would dramatically increase its complexity.

B. Our Proposal

We define now an efficient extension of the ST kernel which, given a well-defined ordering as the one described in Section IV, is not sensitive to the disposition of the nodes with identical discrete labels. The way we propose to extend $C()$ function of the ST kernel to handle complex node labels is the following:

$$C_{\mathrm{CST}}(v_1, v_2) = \begin{cases} \lambda \cdot K_A(v_1, v_2) & \text{if } L(v_1) = L(v_2) \\ & \wedge v_1, v_2 \text{ are leaf nodes} \\ \lambda \cdot K_A(v_1, v_2) \cdot C_{\mathrm{ST}}(v_1, v_2) & \\ & \text{if } L(v_1) = L(v_2) \wedge \ \rho(v_1) = \rho(v_2) \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

Note that the rightmost quantity on the right-hand side of case 2 in the equation is the original C_{ST} function in (3), i.e., it does not consider continuous labels. Thus, the kernel for complex node labels is applied only to the root node of the tree-features. However, if $C_{\mathrm{ST}}(v_1, v_2) > 0$ then $C_{\mathrm{ST}}(v'_1, v'_2) > 0$ for all the pairs of nodes (v'_1, v'_2) for which $C_{\mathrm{ST}}()$ is recursively evaluated during the computation of $C_{\mathrm{ST}}(v_1, v_2)$. As a consequence, recalling that (5) is used in combination with (2), when $C_{\mathrm{CST}}(v_1, v_2)$ is evaluated, the kernel $K_A()$ is surely evaluated as well. Thus, the continuous labels belonging to all the nodes of a matching tree-feature contribute to the determination of the kernel value. Since K_A is only evaluated on root nodes and (5) is evaluated on all pairs of nodes of the input graphs, the outcome of the kernel is clearly independent of the disposition of (sibling) nodes with identical discrete labels. Equation (5) defines a positive semidefinite kernel, since it is the product of positive semidefinite kernels. By using the well-defined ordering in (1), (2), instantiated with (5), is a valid kernel as showed in [4]. We call it ODDCL$_{\mathrm{ST}}$ kernel. In the following, we instantiate the kernel on vectorial attributes as the Gaussian kernel: $K_A(v, v') = e^{-\beta \|A(v) - A(v')\|^2}$ (here β is a kernel parameter) since that is kernel, we will use in Section VI.

Equation (5) has the only purpose to show how the computation of the ST kernel changes from the discrete to the complex node label domain. The algorithm we are going to use to compute the kernel is more efficient than the direct evaluation of (5), and it is based on a fast algorithm for computing the ST kernel for discrete node labels [17]. The kernel has been implemented in Python and the algorithms presented in this section are (simplified) snippets of the actual code.

Algorithm 1: Sketch of an Algorithm to Compute the FeatureMap of a Graph. The Notation Is Python-Style: {} Is an HashMap, and the **in** Operator Applied to an HashMap Performs the Lookup of the Element in It

```
1  def computeFeatureMap(G,h)
       Data: G= a graph
       Data: h= maximum depth of the considered structures
       Result: FeatureMap={subtreeID:{veclabels:freq}}
       Result: SizeMap={subtreeID:size}
2      DDs=computeDecompositionDAGs(G,h);
3      ODDs=order(DDs);
4      FeatureMap={};
5      for ODD in ODDs do
6          for v in topologicalSort(ODD) do
7              for j in 0...h do
8                  subtreeID=encode(T_j(v));
9                  SizeMap[subtreeID]=|T_j(v)|;
10                 if subtreeID not in FeatureMap then
11                     FeatureMap[subtreeID]={v:1};
12                 else
13                     if v not in FeatureMap[subtreeID] then
14                         FeatureMap[subtreeID][v]=1;
15                     else
16                         FeatureMap[subtreeID][v]+=1;
17     return FeatureMap, SizeMap
```

Algorithm 1 computes the *FeatureMap* of a graph G. *FeatureMap* is an HashMap that indicates all the proper subtrees that appear in the graph considering only the discrete node labels. Each subtree encoded by a value x in the FeatureMap, has associated another HashMap containing all the continuous attribute vectors of each subtree encoded by the same value x in the FeatureMap. Note that, in our implementation, FeatureMap is a Python dictionary indicated by strings that uniquely encode trees (function *encode* in line 8 of Algorithm 1). We recall that the subtree features do not consider the

Authorized licensed use limited to: POLO BIBLIOTECARIO DI INGEGNERIA. Downloaded on June 03,2020 at 07:28:49 UTC from IEEE Xplore. Restrictions apply.

continuous labels, thus the same subtree may appear multiple times in the same graph. Each attribute vector has associated its frequency.

Only subtrees with identical structures and discrete labels are encoded by the same hash value. The value of $C_{\mathrm{ST}}()$ for two subtrees encoded by the same hash value is $C_{\mathrm{ST}}(v_1, v_2) = \lambda^{|T(v_1)|}$ [17], thus we only need to know the size of the subtree. Let us now analyze the computational complexity of the algorithm. Lines 2 and 3 require $O(mn)$ and $O(nm \log \rho)$ time, respectively (see Section III). Since the nodes are sorted in inverse topological order, the encoding of line 8 can be computed with a time complexity of $O(\rho)$ [4]. Let $H \leq \lfloor (\rho^{h+1} + 1/\rho - 1) \rfloor \leq n$ be the average number of nodes in a DD (equivalently, nH is the notal number of nodes in all the DDs). Lines 8–16 insert a single feature in the $FeatureMap$. The total number of features generated from a graph G is then $nH(h+1) \leq (h+1)n^2$ (lines 5–7). The (amortized) cost of inserting all such features in the FeatureMap is $O(nHh)$. The overall computational complexity of Algorithm 1 is then $O(n(m \log \rho + hH))$. Note that Algorithm 1 has to be executed only once per example.

Algorithm 2: Sketch of an Algorithm for Computing the $\mathrm{ODDCL}_{\mathrm{ST}}$ Kernel. λ Is a Kernel Parameter

```
1  def ODDCLkernel(G1,G2,λ,h)
       Data: G1,G2=two graphs; λ= a weighting parameter
       Result: k=the kernel value
2      FM1,SM1=computeFeatureMap(G1,h);
3      FM2,SM2=computeFeatureMap(G2,h);
4      k=0;
5      for subtreeID in FM1 do
6          if subtreeID in FM2 then
               /* subtreeID is a feature generated
                  from vertices v_i ∈ G1 and v_j ∈ G2
                  where C_ST(v_i, v_j) ≠ 0                */
7              for v_i in FM1[subtreeID] do
8                  for v_j in FM2[subtreeID] do
9                      freq1=FM1[subtreeID][v_i];
10                     freq2=FM2[subtreeID][v_j];
11                     size=SM1[subtreeID];
12                     k+=freq1·freq2·λ^size·K_A(v_i,v_j)
                           /* C_ST(v_i, v_j) = λ^size      */
13     return k
```

Algorithm 2 sketches the code to compute the kernel value between two graphs. Once the FeatureMaps have been computed with Algorithm 1, to calculate the kernel we need to search for matching subtree features in the two FeatureMaps. For any matching subtree feature *subtreeID*, we need to compute the kernel $K_A(v_i, v_j), v_i \in V_{G_1}, v_j \in V_{G_2}$ for each pair of vertices that generate the feature *subtreeID* in the two graphs. The complexity of Algorithm 2 is linear in the number of discrete features (lines 5 and 6) and quadratic in the lists of vectorial labels associated with each discrete feature. Note that there are at most n different attribute vectors in the original graph (one associated with each node), so each discrete feature can be associated with at the most n different vectorial labels. Thus, the distribution of $O(hHn)$ elements in a FeatureMap (FM1 or FM2) that maximizes the computational complexity is the one having $O(hH)$ different discrete features, each one with an associated list of vectorial labels of size $O(n)$. The complexity of Algorithm 2 is then $O(hHn^2\ Q(K_A))$.

Note that computing (4) with Algorithm 2 would not be feasible: the kernel $K_A(v_i, v_j)$ computed on line 12, which is now computed on a single pair of nodes, should be computed on the whole set of nodes composing the subtrees rooted at v_i, v_j.

C. Computation Speed Up With RBF Kernel Approximation

Profiling the execution of Algorithm 2, the most expensive step is the computation of $K_A()$ for all the pairs of vectorial labels associated with a subtree feature (lines 7–12). In order to speed up the kernel computation, we propose to approximate this step. Recently, [18] proposed a method to generate an (approximated) explicit feature space representation for the RBF kernel by Monte Carlo approximation of its Fourier transform. This procedure depends on a parameter D determining the size of the approximated feature vector. In the following, we refer to the approximated feature vector of a node v as $\hat{\phi}^D_{\mathrm{RBF}}(v)$; the kernel induced by $\hat{\phi}^D_{\mathrm{RBF}}()$ is positive semidefinite [18]. Note that other approximations, such as Nyström [19], can be used. Assuming we have one set of identical (with respect to discrete labels only) proper subtrees V_1 related to a graph G_1, and a second set V_2 related to a graph G_2, we can approximate the computation of the kernel $K_A()$ between the all pairs ($v_i \in V_1\ and v_j \in V_2$) as

$$\sum_{v_i \in V_1} \sum_{v_j \in V_2} k_{\mathrm{RBF}}(v_i, v_j) \simeq \left\langle \sum_{v_i} \hat{\phi}^D_{\mathrm{RBF}}(v_i), \sum_{v_j} \hat{\phi}^D_{\mathrm{RBF}}(v_j) \right\rangle. \quad (6)$$

We can now substitute lines 11–17 of Algorithm 1 in order to associate to subtreeID just the sum of the explicit RBF vectors generated from the vectorial labels. The resulting complexity of Algorithm 1 is $O(n(m \log \rho + \mathrm{hHD})$, while the complexity of Algorithm 2 drops to $O(\mathrm{nhHD})$. Note that, in practice, the computational gain due to approximation might be higher than what the worst case analysis suggests (see Section VI), because the worst case scenarios of the two versions of the algorithm are different. In the approximated case, the complexity is independent of the number of continuous attributes associated with a discrete feature, and the worst case is the one that maximizes the number of discrete features.

VI. Experiments

In this section, we compare the $\mathrm{ODDCL}_{\mathrm{ST}}$ kernel presented in Section V and its approximated version presented in Section V-C with several state-of-the-art kernels for graphs with continuous labels. After the description of the experimental setup in Section VI-A, we discuss the predictive performance of the different kernels in Section VI-B. In Section VI-C, we compare the computational times required by the different kernel calculations.

A. Experimental Setup

The experimental setup follows the one in [10]: the results presented in this section refer to an Support Vector Machine (SVM) classifier[1] in a process of nested ten-fold cross validation, in which the kernel (and the classifier) parameters are validated using the training data set only; the experiments have been repeated ten times (with different cross validation splits), and the average accuracy results with standard deviation are reported. The proposed kernel parameter values have been cross validated from the following sets: $h = \{0, 1, 2, 3\}$ and $\lambda = \{0.1, 0.3, 0.5, 0.7, 1, 1.2\}$. For $\mathrm{ODDCL}_{\mathrm{Approx}}$ D parameter has been selected after a preliminary experiment: a nested ten-fold cross validation on the Enzymes data set was performed using the default values for all other parameters and varying only D; $D = 1000$ looked like a good tradeoff between speed and accuracy. Note that, with very high values of D, it is expected to reproduce almost exactly the results of the $\mathrm{ODDCL}_{\mathrm{ST}}$ kernel. The SVM C parameter has been selected

[1] https://github.com/nickgentoo/scikit-learn-graph

Authorized licensed use limited to: POLO BIBLIOTECARIO DI INGEGNERIA. Downloaded on June 03,2020 at 07:28:49 UTC from IEEE Xplore. Restrictions apply.

TABLE I

AVERAGE ACCURACY RESULTS ± STANDARD DEVIATION IN NESTED TEN-FOLD CROSS VALIDATION OF THE PROPOSED ODDCL$_{ST}$, GRAPHHOPPER, SHORTEST PATH, COMMON SUBGRAPH MATCHING, PROPAGATION, GRAPH INVARIANT, WEISFEILER–LEHMAN, AND ODD$_{ST}$ KERNELS. THE LAST TWO KERNELS DO NOT CONSIDER INFORMATION FROM NODE ATTRIBUTES. THE BEST ACCURACY FOR EACH DATA SET IS REPORTED IN BOLD

$Kernel$	COX2	BZR	DHFR	ENZYMES	PROTEINS	SYNTH
ODDCL$_{ST}$	**83.01±1.95**	**92.85±0.30**	**84.05±3.02**	**73.40±0.98**	75.86 ± 0.77	95.83± 0.73
ODDCL$_{Approx}$	79.99±1.20	87.12±1.14	81.48±0.36	69.95 ± 1.12	75.93 ± 0.59	96.1 ± 0.74
GH	-	-	-	68.3±1.31	74.1±0.5	78.3±1.5
SP	-	-	-	72.3±0.9	75.5±0.8	82.5±1.3
CSM	-	-	-	69.5±0.7	out of memory	out of time
P2K	77.41±0.65	82.06±0.51	67.23±0.84	67.99	69.37±0.81	75.16±1.37
GIK_NSPDK	71.68±0.99	86.84±0.55	70.76±1.07	70.78±0.91	76.18±0.52	83.46±1.38
GIK_SGK_GLOBAL	78.09±0.58	70.80±0.96	81.41±0.65	71.38±0.53	76.53±0.53	83.43±1.41
GIK_SGK_LOCAL	81.07±0.82	86.23±0.67	81.24±0.66	71.40±0.60	76.22±0.40	83.83±1.31
GIK_WL_GLOBAL	81.63±0.98	69.11±0.93	81.46±0.84	70.21±0.88	76.17±0.48	82.66±1.02
GIK_WL_LOCAL	71.66±0.97	86.54±0.85	71.66±0.97	71.85±0.71	**76.62±0.63**	82.60±0.77
WL	81.45±0.76	88.41±0.42	82.76±0.41	48.5±0.7	75.6±0.5	**97.5±2.4**
ODD$_{ST}$	82.76±0.97	88.21 ± 0.54	83.75 ± 1.03	51.26±1.03	73.85±0.55	96.63±0.64

in the set $\{0.01, 0.1, 1, 10, 100, 1000, 10\,000\}$. All the considered kernels depend on node kernels for continuous vectors and/or discrete node labels. The kernel for continuous attributes has been fixed for all the kernels as $K_A(v_1, v_2) = e^{-\lambda\|A(v_1)-A(v_2)\|^2}$ with $\lambda = 1/d$ and d the size of the vector of attributes $A(v_i)$. Where discrete labels were not available, the degree of each node has been considered as the node label. Note that this step is not strictly necessary, since if the graphs have no discrete label information, we can just assume all the nodes having the same label. However, agreement on out-degree is an effective way to speed up computation and increase the discriminativeness of kernels. Following [10], the kernel matrices have been normalized.

We tested our method on the (publicly available) data sets from [10]: *ENZYMES*, *PROTEINS* and *SYNTHETIC*, and from [11]: COX2, BZR, DHFR. *ENZYMES* (symmetrized version) is a set of proteins from the BRENDA database [20]. Each protein is represented as a graph, where nodes correspond to secondary structure elements. Two nodes are connected whenever they are neighbors either in the amino acid sequence or in the 3-D space of the protein tertiary structure [1]. Each node has a discrete attribute indicating the structure it belongs to (helix, sheet, or turn). Moreover, several chemical and physical measurements can be associated with each node, obtaining a vector-valued attribute associated with each node. Examples of these measurements are the length of the secondary structure element in Å, its hydrophobicity, polarity, polarizability, and so on. The task is the classification of enzymes into one out of six EC top-level classes. There are 100 graphs per class in the data set. The average number of nodes and edges of the graphs is 32.6 and 46.7, respectively. The size of the vectors associated with the nodes is 18.

PROTEINS is the data set from [21]. The proteins are represented as graphs as described earlier. The task is to distinguish between enzymes and nonenzymes. There are 1113 graphs in the data set, each one with an average of 39.1 nodes and 72.8 edges. The dimension of the continuous node attributes is one.

SYNTHETIC is a data set presented in [10]. A random graph with 100 nodes and 196 edges has been generated. Each node has a corresponding continuous label sampled from $\mathcal{N}(0, 1)$. Then two classes of graphs have been generated with 150 graphs each. Each graph in the first class was generated rewiring five edges and permuting ten node attributes from the original graph, while for each graph in the second class ten edges and five node attributes has been modified. Finally, noise from $\mathcal{N}(0, 0.45^2)$ has been added to each node attribute in both classes. *COX2, BZR* and *DHFR*, originally presented in [22] are data sets of chemical compounds where the target is to predict their toxicity. The 3-D coordinates of the atoms have been used as node attributes. COX2 counts 467 graphs, while BZR and DHFR 405 and 756, respectively. The average number of nodes for the three data sets is 41.2, 35.7, and 42.4, while the average number of edges is 43.44, 38.35, and 44.5, respectively.

B. Experimental Results

In Table I, we report the experimental results of the proposed ODDCL$_{ST}$ kernel, its approximated version ODDCL$_{Approx}$ presented in Section V-C, the Propagation kernel ($P2K$) [11], different instantiations of the Graph Invariant Kernels [12] (GIK_NSPDK, GIK_SGK_GLOBAL, GIK_SGK_LOCAL, GIK_WL_GLOBAL, and GIK_WL_LOCAL) and the results from the paper [10], corrected according to the *erratum*. All these kernels can deal with continuous labels. For sake of comparison, the results of the Weisfeiler–Lehman kernel (WL) [7] and ODD$_{ST}$ [4] kernels that can deal only with discrete attributes, are reported too.

In the *ENZYMES* data set, the proposed ODDCL$_{ST}$ kernel is the best performing one, while its approximated version performs better than GraphHopper kernel (GH), Shortest Path kernel (SP), Connected Subgraph Matching kernel (CSM) and P2K. This gives evidence that our proposed kernels are able to extract useful information from nondiscrete labels in a more effective way than the other kernels. Note, however, that the computational requirements of SP are prohibitive on this data set, as it will be detailed later in this section. In this data set, the WL and the ODD$_{ST}$ kernels perform poorly, indicating that the information encoded by continuous attributes is relevant to the task. In *PROTEINS*, GIK kernels are the best performing ones, with ODDCL$_{ST}$ and ODDCL$_{Approx}$ kernels showing comparable performances. Before analyzing the results for the *SYNTHETIC* data set, we need to draw some considerations. The first reported results on this data set were affected by a bug. The correct results have been published later, and depicted a scenario where the kernels that did not consider continuous labels performed better than the others. However, it is interesting to evaluate the performance of the different kernels on this data set, because it shows how much a kernel is resistant to noise. In the *SYNTHETIC* data set, even if the proposed kernels perform better than other competitors that consider continuous attributes, its performance are not able to achieve the WL and ODD$_{ST}$ ones. This means that, for this data set, the information provided by vectorial labels is basically noise. This fact is more evident when looking at the performance of the other kernels that consider vectorial labels. However, it is clear that the proposed

Authorized licensed use limited to: POLO BIBLIOTECARIO DI INGEGNERIA. Downloaded on June 03,2020 at 07:28:49 UTC from IEEE Xplore. Restrictions apply.

TABLE II

TIME REQUIRED FOR THE KERNEL MATRIX COMPUTATION OF ODDCL_{ST}, $\text{ODDCL}_{\text{Approx}}$, GRAPHHOPPER, SHORTEST PATH, PROPAGATION, AND GRAPH INVARIANT KERNELS.$*$: TIMES REFERRING TO GH AND SP ARE REPORTED FROM [10]. NOTATION d: DAY(S); h: HOUR(S), m: MINUTE(S); s: SECOND (S)

Kernel	ENZYMES	PROTEINS	SYNTH
ODDCL_{ST}	35m	21m	23m
ODDCL_{Approx}	1.8m	5.2m	7.5m
GH*	12m10s	2.8h	12m10s
SP*	3d	7.7d	3.4d
GIKs (average)	48m	3h49m	16.3m
P2K	6s	28s	5s

ODDCL_{ST} kernel and its approximated version are more tolerant to this kind of noise than competing kernels. Moreover, for the proposed kernel, it is possible to consider an additional parameter, to be cross-validated with the others, that indicates if the vectorial labels have to be considered or not. We recall that, without considering vectorial labels, ODDCL_{ST} kernel reduces to the ODD_{ST} kernel.

On COX2 and DHFR data sets, ODDCL_{ST} is the best performing kernel, while its approximated version is competitive with the other kernels. On BZR data set, ODDCL_{ST} and $\text{ODDCL}_{\text{Approx}}$ are the first and the second best performing kernels among the ones considering vectorial attributes.

Summarizing the results, ODDCL_{ST} kernel performs better than the other kernels that consider vectorial labels on five out of six data sets. At the same time, $\text{ODDCL}_{\text{Approx}}$ is competitive with the other kernels; in particular, it performs always better than GH (where applicable) and P2K.

In order to assess the differences in performance among the kernels, we performed the Friedman test with the Nemenyi posthoc test ($\alpha = 0.05$) [23]. Since not all results are available for all kernels on all data sets, we performed one test on all data sets, but excluding the kernels GH, SP, CSM; a second test includes all kernels, but it is performed on ENZYMES, PROTEINS, SYNTHETIC data sets: ODDCL_{ST} is significantly better than all other kernels except GIK_SGK_LOCAL, $\text{ODDCL}_{\text{Approx}}$, ODD_{ST}, and WL. Since WL does not consider continuous labels, we performed a further analysis focusing on the data sets for which such information is important: COX2, BZR, DHFR, and ENZYMES. On such data sets ODDCL_{ST} always outperforms WL; we performed a t-test [23], which showed that ODDCL_{ST} is significantly better than WL at level $\alpha = 0.05$ on BZR and ENZYMES.

C. Computational Times

The computational times reported in this section refer to the Gram matrix computation of each kernel on the three largest data sets. Since the kernels are implemented in different languages, we considered for the sake of comparison the computational times reported in [10] for GH and SP kernels. We want to point out that the times reported in this section that have to be considered just as orders of magnitude. Table II reports such computational times for the different kernels. P2K is the fastest kernel; however, its predictive performances are the poorest. The proposed ODDCL_{ST} kernel is faster than the SP kernel in all the considered data sets. With respect to GraphHopper, ODDCL_{ST} is slightly slower on ENZYMES and SYNTH (roughly two times slower, but still in the order of min), while it is considerably faster on the PROTEINS data set (eight times faster). The computational requirements of ODDCL_{ST} and GIKs are comparable on ENZYMES and SYNTH data sets, while ODDCL_{ST} is considerably faster on the PROTEINS data set (almost 11 times faster). Let us now consider the approximated version of our proposed kernel, $\text{ODDCL}_{\text{Approx}}$. It is the second fastest kernel (after P2K), with a significant difference with respect to GraphHopper while being more accurate in all the considered data sets. $\text{ODDCL}_{\text{Approx}}$ is also considerably faster than GIKs, being more accurate in two out of three of the considered data sets. Interestingly, because of the different substructures considered by the two kernels, GraphHopper kernel have the lowest run-times in ENZYMES and SYNTHETIC, while for ODDCL_{ST} PROTEINS is the data set that requires the lowest computational resources. We can argue that this happens because of the higher number of edges of proteins, that directly influence the sparsity of ODDCL_{ST} kernel. On the other hand, $\text{ODDCL}_{\text{Approx}}$ shows the maximum speedup on the ENZYMES data set. The speedup obtained by the approximated version is proportional to the number of different discrete features generated by the ODD base kernel, and thus is influenced by the number of different discrete labels in the data set, in addition to the graph topology.

VII. CONCLUSION

In this brief, we have presented an extension to continuous attributes of the ODD kernel framework for graphs. Moreover, we have studied the performances of a continuous attributes graph kernel derived by the ST kernel for trees. Experimental results on reference data sets show that the resulting kernel is both fast to compute and quite effective on all studied data sets, which is not the case for continuous attributes graph kernels presented in the literature. A faster but approximated version of the proposed kernel, returning satisfying predictive performances, is presented as well.

REFERENCES

[1] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics*, vol. 21, pp. 47–56, Jun. 2005.

[2] S. Karaman, L. Seidenari, S. Ma, A. Del Bimbo, and S. Sclaroff, "Adaptive structured pooling for action recognition," in *Proc. BMVA*, 2014, pp. 1–12.

[3] F. Costa and K. De Grave, "Fast neighborhood subgraph pairwise distance kernel," in *Proc. Int. Conf. Mach. Learn.*, Haifa, Israel, 21-24 June 2010, pp. 255–262.

[4] G. Da San Martino, N. Navarin, and A. Sperduti, "A tree-based kernel for graphs," in *Proc. 12th SIAM Int. Conf. Data Mining*, 2012, pp. 975–986.

[5] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," in *Proc. 16th Annu. Conf. Comput. Learn. Theory 7th Kernel Workshop*, vol. 2777, 2003, pp. 129–143.

[6] F. Costa and K. De Grave, "Fast neighborhood subgraph pairwise distance kernel," in *Proc. ICML*, 2010, pp. 255–262.

[7] N. Shervashidze and K. M. Borgwardt, "Fast subtree kernels on graphs," in *Proc. NIPS*, 2009, pp. 1660–1668.

[8] M. Neumann, N. Patricia, R. Garnett, and K. Kersting, "Efficient graph kernels by randomization," in *Proc. ECML PKDD*, vol. 7523, Berlin, Germany, 2012, pp. 378–393.

[9] N. Kriege and P. Mutzel, "Subgraph matching kernels for attributed graphs," in *Proc. ICML*, 2012, pp. 1015–1022.

[10] A. Feragen, N. Kasenburg, J. Petersen, M. de Bruijne, and K. M. Borgwardt, "Scalable kernels for graphs with continuous attributes," in *Proc. NIPS*, 2013, pp. 216–224.

[11] M. Neumann, R. Garnett, C. Bauckhage, and K. Kersting, "Propagation kernels: Efficient graph kernels from propagated information," *Mach. Learn.*, vol. 102, no. 2, pp. 209–245, Feb. 2015.

[12] F. Orsini, P. Frasconi, and L. De Raedt, "Graph invariant kernels," in *Proc. IJCAI*, 2015, pp. 3756–3762.

[13] L. Oneto, N. Navarin, M. Donini, A. Sperduti, F. Aiolli, and D. Anguita, "Measuring the expressivity of graph kernels through statistical learning theory," *Neurocomputing*, Apr. 2017. [Online]. Available: https://doi.org/10.1016/j.neucom.2017.02.088

[14] F. Aiolli, M. Donini, N. Navarin, and A. Sperduti, "Multiple graph-kernel learning," in *Proc. IEEE Symp. Comput. Intell. Data Mining*. Cape Town, Dec. 2015, pp. 1607–1614.

[15] G. Da San Martino, N. Navarin, and A. Sperduti, "Ordered decompositional DAG kernels enhancements," *Neurocomputing*, vol. 192, pp. 92–103, Jun. 2016.

Authorized licensed use limited to: POLO BIBLIOTECARIO DI INGEGNERIA. Downloaded on June 03,2020 at 07:28:49 UTC from IEEE Xplore. Restrictions apply.

[16] A. Moschitti, "Making tree kernels practical for natural language learning," in *Proc. 11th Conf. Eur. Chapter Assoc. Comput. Linguistics*, 2006, pp. 113–120.
[17] S. V. N. Vishwanathan and A. J. Smola, "Fast kernels for string and tree matching," in *Proc. NIPS*, 2002, pp. 569–576.
[18] A. Rahimi and B. Recht, "Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning," in *Proc. NIPS*, 2009, pp. 1313–1320.
[19] C. Williams and M. Seeger, "Using the Nyström method to speed up kernel machines," in *Proc. Adv. Neural Inf. Process. Syst.*, 2001, pp. 682–688. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.7519
[20] I. Schomburg *et al.*, "BRENDA, the enzyme database: Updates and major new developments," *Nucl. Acids Res.*, vol. 32, pp. D431–D433, Jan. 2004.
[21] P. D. Dobson and A. J. Doig, "Distinguishing enzyme structures from non-enzymes without alignments," *J. Mol. Biol.*, vol. 330, no. 4, pp. 771–783, Jul. 2003.
[22] P. Mahé and J.-P. Vert, "Graph kernels based on tree patterns for molecules," *Mach. Learn.*, vol. 75, no. 1, pp. 3–35, Apr. 2009.
[23] N. Japkowicz and M. Shah, *Evaluating Learning Algorithms: A Classification Perspective*. New York, NY, USA: Cambridge Univ. Press, 2011.

Authorized licensed use limited to: POLO BIBLIOTECARIO DI INGEGNERIA. Downloaded on June 03,2020 at 07:28:49 UTC from IEEE Xplore. Restrictions apply.