Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

Ordered Decompositional DAG kernels enhancements

Giovanni Da San Martino [a], Nicolò Navarin [b,*], Alessandro Sperduti [b]

[a] Qatar Computing Research Institute, HBKU, P.O. Box 5825, Doha, Qatar
[b] Department of Mathematics, University of Padova, via Trieste 63, Padova, Italy

ARTICLE INFO

Article history:
Received 12 July 2015
Received in revised form
23 December 2015
Accepted 27 December 2015
Available online 27 February 2016

Keywords:
Kernel methods
Kernel functions
Graph kernels
Classification

ABSTRACT

In this paper, we show how the Ordered Decomposition DAGs (ODD) kernel framework, a framework that allows the definition of graph kernels from tree kernels, allows to easily define new state-of-the-art graph kernels. Here we consider a fast graph kernel based on the Subtree kernel (ST), and we propose various enhancements to increase its expressiveness. The proposed DAG kernel has the same worst-case complexity as the one based on ST, but an improved expressivity due to an augmented set of features. Moreover, we propose a novel weighting scheme for the features, which can be applied to other kernels of the ODD framework. These improvements allow the proposed kernels to improve on the classification performances of the ST-based kernel for several real-world datasets, reaching state-of-the-art performances.

1. Introduction

The increasing availability of data in structured form, such as trees [1] or graphs [2–4], has led to the development of machine learning techniques able to deal directly with such types of data. Among these, kernel methods, such as Support Vector Machines (SVM) [5], have become very popular due to their generalization ability and state of the art performances in many tasks, such as relationship extraction [6], analysis of RDF data [7], action recognition [8], text categorization of biomedical data [9] and bioinformatics [10].

The class of kernel methods comprises all those learning algorithms which do not require an explicit representation of the input, but only information about the similarities among them. A simple way of assessing the similarity between two objects described by a set of features is to compute the dot product of their representation in feature space. If a "similarity" function $k(\cdot,\cdot)$, corresponding to a dot product $\langle\cdot,\cdot\rangle$ in feature space, is available, the intermediate step of explicitly representing the data can be avoided. In fact, computing $k(x_1,x_2)$ implicitly corresponds to perform a nonlinear transformation of the input vectors x_1 and x_2 via a function $\phi(\cdot)$ and then to compute the dot product of the resulting vectors $\phi(x_1)$ and $\phi(x_2)$. The function $\phi(\cdot)$ projects the input vectors into a feature space of much higher (possibly infinite) dimension where it is more likely to accomplish the learning task. Kernel methods generally formulate a learning problem as a constrained optimization one, where an objective function combining an empirical risk term with a (quadratic) regularizer must be minimized. If the employed kernel function is symmetric positive semidefinite, the problem is convex and thus has a global minimum [5].

Any kernel method can be decomposed into two modules: (i) a problem specific kernel function; (ii) a general purpose learning algorithm (the solver). Since the solver interfaces with the problem only by means of the kernel function, it can be used with any kernel function, and vice versa. Examples of popular kernel methods are the perceptron [11] for the on-line setting, and the Support Vector Machines [5] for the batch setting. Note that, provided an appropriate kernel function is given, any kernel method can be applied to any type of data. More importantly, the kernel function encodes all the information about the input data, thus the definition of appropriate kernel functions is crucial for the outcome of the learning algorithm.

A popular strategy for defining kernel functions for structured data is to decompose the structures into their constituent parts, and then, for each pair of parts, apply a local kernel [12]. While this strategy has been proved successful for strings and trees [13–18], it is not directly applicable to graphs because of the computational complexity issues which arise: representing a graph in terms of its subgraphs is not feasible since subgraph isomorphism, an *NP-complete* problem, should be solved for each pair of subgraphs. In [19] it has been demonstrated that any kernel whose feature space mapping is injective is as hard to compute as graph isomorphism, an *NP* problem that still is not known whether it is in *P* or if it is *NP-complete*. Due to this limitation, the available strategies for

* Corresponding author.
E-mail addresses: gmartino@qf.org.qa (G. Da San Martino), nnavarin@math.unipd.it (N. Navarin), sperduti@math.unipd.it (A. Sperduti).

http://dx.doi.org/10.1016/j.neucom.2015.12.110
0925-2312/© 2016 Elsevier B.V. All rights reserved.

building kernels are: (i) restricting the input domain to a class of graphs for which isomorphism can be checked quickly [20]; (ii) select a priori a set of features, usually corresponding to a specific type of substructure, such as walks [19], paths [21,22], subtree patterns [23,24]. The former approach can be applied to a limited type of graphs, the latter tends to have a limited flexibility: when the available kernels are not relevant to the task, a new one has to be designed. However, defining an efficient symmetric positive semidefinite kernel, corresponding to the desired feature space, can be an extremely difficult task. All the above approaches discard information about the original graph and are effective only when the selected features are relevant for the current problem. We propose to design graph kernels as follows: first transform the graphs into simpler structures, i.e. multisets of directed acyclic graphs (DAGs), and then extend the definition of a large class of already available kernels for trees to DAGs. Our approach allows the application of the vast literature on kernels for trees, which consists of fast and/or very expressive kernels, to the graph domain.

Generally speaking, a serious drawback which prevents many of the kernels listed above to be applied to large datasets is their computational time complexity. Those kernels which can be applied to large datasets exploit a “limited” number of features to represent a graph. For example, the kernel proposed in [24] has a linear complexity in the number of edges of the graphs because any graph is represented in the feature space by a number of non-zero features which is proportional to the number of nodes of the graph. On the other hand, a too compact representation of a graph in feature space may have a negative impact on the effectiveness of the kernel because of a reduced discrimination ability.

In this paper, we tackle this problem by proposing various enhancements to a fast graph kernel based on the Subtree kernel for trees (ST) [25]. Among these, the main contribution is a novel tree kernel, which has the same worst-case complexity of the ST kernel, while the size of its feature space is much larger.

The paper is structured as follows. Section 2 introduces some basic notation and definitions. Section 3 recalls the ODD framework, of which the proposed kernels are instances. Section 4 describes the main contributions of the paper: the ST+kernel for DAGs and a novel weighting scheme for the features, which can be applied to other kernels of the ODD framework. Section 5 discusses some related kernels for graphs, and Section 6 provides experimental evidence of the effectiveness of the proposed approaches. Finally, Section 7 draws conclusions.

The paper extends the work in [26] by adding: (i) a self-contained and simplified description of the ST+kernel; (ii) a novel, more effective, feature weighting scheme; (iii) an extended and revised “Related Work” section; (iv) a novel set of experiments which are now performed on much larger benchmark datasets and for a larger number of competing graph kernels; (v) a comparison among empirical execution times of the various experimented kernels.

2. Notation

A graph is a triplet $G=(V,E,L)$, where V (alternatively V_G) is the set of nodes ($|V|$ is the number of nodes), E the set of edges and $L()$ a function returning the label of a node. All labels are obtained from a fixed alphabet $\mathcal{A}$. A graph is undirected if $(v_i,v_j)\in E \Leftrightarrow (v_j,v_i)\in E$, otherwise it is directed. A path in a graph is a sequence of nodes $v_1,\dots,v_n$ such that $v_i \in V, 1\le i\le n$, $(v_i,v_{i+1})\in E$ and $\forall 1\le i\le n, 1\le j<n, j\ne i.\ v_i \ne v_j$ (no node, except the first one, can appear twice in the same path). A cycle is a path for which $v_1=v_n$; a cycle is even/odd if its number of nodes is even/odd, respectively. A graph is connected if there exists a path connecting each pair of nodes. A connected graph is rooted if exactly one node has no incoming edges. A graph is ordered if the set of neighbors of each node is ordered. A tree is a rooted connected directed acyclic graph where each node has at most one incoming edge. A subtree of a tree T is a connected subset of nodes of T. A proper subtree is a subtree composed of a node and all of its descendants. Given a node v of a tree, $\rho(v)$ represents the outdegree of v, i.e. the number of nodes connected to v. We will use ρ as the maximum outdegree of a node in either a tree or a graph. The depth $depth(v)$ of a node v is the number of edges in the shortest path between the root of the tree and v. If the tree is ordered, $ch_v[j]$ represents the j-th child of v and $chs_v[j_1,j_2,\dots,j_n]$ indicates the set of children of v with indices j_1, $j_2,\dots,j_n$. Given a graph G and a node $v\in V(G)$, we define a subtree-walk of size h as the tree obtained by the following procedure: the root of the tree is v; at each step i, with $1\le i\le h$, and for each current leaf node v_j of the tree, any neighboring node of v_j in G is added to the tree as a child of v_j. Note that, when $h>1$, typically a node of G can appear multiple times in the same subtree-walk. Given a DAG D and a node $v_i\in V(D)$, we define a tree-visit, denoted by $\overset{v_i}{\Delta}$, as the tree resulting from the visit of D starting from the node v_i. Such visit returns all the nodes of D reachable from v_i. If a node v_j can be reached more than once, more occurrences of v_j will appear in $\overset{v_i}{\Delta}$ (see Fig. 2b for an example).

3. Preprocessing: from graphs to multisets of DAGs

This section recalls the ODD-Kernels framework for graphs [27]. The idea of our approach is to transform the graphs into simpler structures, i.e. DAGs, and then apply a kernel for such structures. The following subsections explain each step of the transformation.

3.1. From graph to DAGs

The graph G is mapped into a multiset of DAGs $DD_G=\{DD_G^{v_i}\,|\,v_i\in V_G\}$, where $DD_G^{v_i}=(V_G^{v_i},E_G^{v_i},L)$ is obtained by keeping each edge in the shortest path(s) connecting v_i with any $v_j\in V_G$. From a practical point of view, $DD_G^{v_i}$ can be built by performing a breadth-first visit on the graph G starting from node v_i and applying the following rules:

1. during the visit a direction is given to each edge; if v_j is reached from v_i in one step, then $(v_i,v_j)\in E_G^{v_i}$ (note that edge (v_j,v_i) is not added to $E_G^{v_i}$);
2. edges connecting nodes reached at level l of the visit to nodes reached at level $g<l$ are not added to $E_G^{v_i}$ (such edges would induce a cycle in $DD_G^{v_i}$).

For every choice of G and v_i, a single *Decompositional Dag* $DD_G^{v_i}$ is generated. By repeating the procedure for each node of the graph, $|V|$ DAGs are obtained. Fig. 1 shows the four DDs obtained from the undirected graph in Fig. 1a. Note that when the same node is reached simultaneously (at the same level of the visit) from different nodes, then all involved edges are preserved. For example, when considering the visit at level 2 starting from node **s**, the node **d** is reached simultaneously by edges (**b**, **d**) and (**e**, **d**), and both of them are preserved in the corresponding Decompositional DAG (see Fig. 1b). In order to reduce the total number of nodes of ${DD_G}^v$, we propose to limit the depth of the visits during the generation of the multiset of DAGs [27] to a constant value h. The resulting DAG will be referred to as $DD_G^{v,h}$. Given $v\in V_G$, let H be the number of nodes generated by the visits up to depth h. An upper bound for H is ρ^h. Notice, this is a loose bound, in many practical cases. The total number of nodes of DD_G is $|V_G|H$. Note that, if ρ is constant, then also H is constant.