Substructure Counting Graph Kernels for Machine Learning from RDF Data

Gerben Klaas Dirk de Vries\textsuperscript{a,*}, Steven de Rooij\textsuperscript{b,a}

\textsuperscript{a}System and Network Engineering group
Informatics Institute, University of Amsterdam
Science Park 904, 1098 XH Amsterdam, The Netherlands

\textsuperscript{b}Knowledge Representation and Reasoning group
Department of Computer Science, VU University Amsterdam
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

Abstract
In this paper we introduce a framework for learning from RDF data using graph kernels that count substructures in RDF graphs, which systematically covers most of the existing kernels previously defined and provides a number of new variants. Our definitions include fast kernel variants that are computed directly on the RDF graph. To improve the performance of these kernels we detail two strategies. The first strategy involves ignoring the vertex labels that have a low frequency among the instances. Our second strategy is to remove hubs to simplify the RDF graphs. We test our kernels in a number of classification experiments with real-world RDF datasets. Overall the kernels that count subtrees show the best performance. However, they are closely followed by simple bag of labels baseline kernels. The direct kernels substantially decrease computation time, while keeping performance the same. For the walks counting kernel the decrease in computation time of the approximation is so large that it thereby becomes a computationally viable kernel to use. Ignoring low frequency labels improves the performance for all datasets. The hub removal algorithm increases performance on two out of three of our smaller datasets, but has little impact when used on our larger datasets.

Keywords: Graph Kernels, Machine Learning for RDF, Weisfeiler-Lehman, Hub Removal

1. Introduction
In recent years graph kernels have been introduced as a promising method to perform data mining and machine learning on Linked Data and the Semantic
Web. These methods take Resource Description Framework (RDF) data as input.

One main advantage of this approach is that the techniques are therefore very widely applicable to all kinds of Linked Data. Almost no assumptions are made on the semantics of the data and its content, other than that it is in RDF. So, additionally, these methods require very little knowledge of Semantic Web technologies to be employed.

Another advantage is the host of existing machine learning algorithms, called kernel methods [1, 2], that can be used with these graph kernels. The most well known of these algorithms is the Support Vector Machine (SVM) for classification and regression. However, algorithms exist for ranking [3, 4], clustering [5], outlier detection [6], etc., which can all be used directly with these kernels. More recently, interest has increased for large scale linear classification [7] for larger datasets, for which a number of graph kernels can also be used.

In this paper we give a comprehensive overview of graph kernels for learning from RDF data. We introduce a framework for these kernels, which are based on counting different graph substructures, that encompasses most of the graph kernels previously introduced for RDF, but also introduces new variants. The framework includes fast kernel variants that are computed directly on the RDF graph. We also detail the necessary adaptation of the Weisfeiler-Lehman graph kernel [8] needed to compute a number of kernels in our framework. Furthermore, we give two strategies to further improve the machine learning performance with these kernels. The first strategy ignores vertex labels which have a low frequency of occurrence among the instances and the second strategy removes hubs to simplify the RDF graphs. All of our kernels defined in the framework can be used with large scale linear classification methods.

The kernels are studied in a number of classification experiments on different RDF datasets. The goal of these experiments is to study the influence of the different choices for graph kernels defined in our framework. It turns out that kernel performance differs per dataset. Overall, kernels that count subtrees in the graphs are the best choice. However, simple bag of labels baseline kernels also perform well and are significantly cheaper to compute. The strategy to ignore low frequency labels has a positive effect on performance in all tasks, whereas the hub removal strategy only has a positive effect in a number of tasks and has no influence for larger datasets.

The work presented in this paper consolidates and expands our earlier papers on graph kernels for RDF [9, 10] and hub removal [11].

The rest of this paper is structured as follows. We begin with an overview of related work. In Section 3 we introduce our kernel framework and algorithms. Section 4 covers our experiments with these kernels. We end with conclusions and suggestions for future work.

2. Related Work

Graph kernels, such as those introduced in [12, 8, 13], are methods to perform machine learning on graph structured data, using kernel methods [1, 2].
For learning from RDF data, the intersection subtree and intersection graph kernels were introduced in [14]. A fast approximation of the Weisfeiler-Lehman graph kernel [8], specifically designed for RDF was introduced in [9]. In [10] a fast and simple graph kernel, similar to the intersection subtree kernel was defined. In the context of recommender systems using linked data [15] and [16] introduce kernels based on the labels in the neighborhood of a vertex. The current paper defines a framework which systematically covers most of these previously introduced kernels.

There are a number of papers that cover machine learning from Linked Data and the Semantic Web using kernel methods, which focus on the formal/ontological level of the data, and which are therefore less generally applicable. One of the first papers to introduce kernel methods to learn from the Semantic Web was [17]. In that paper, kernels are defined manually using task relevant properties of the instances. Other approaches are based on description logics, such as [18], and inductive logic programming [19].

The specific task of link prediction on RDF graphs is tackled in several papers. One approach is to use matrix factorization [20, 21]. Another, related approach, is to use tensor factorization [22, 23]. Finally, there are also approaches using (tensor) neural networks [24, 25] and very recently Gaussian processes [26]. The graph kernel methods of this paper are applicable to a more wide range of machine learning and data mining tasks than these link prediction methods. For instance, graph kernels can be used in clustering, to predict labels/classes that are not links in the graph.

The framework for graph kernels for RDF that we define in this paper has some similarities to the comparison of propositionalization [27] strategies for Linked Data in [28], in the sense that some of the propositionalizations are similar to the graph substructures that our graph kernels work with. However, they do not use kernel methods and use Linked Data as background knowledge to enhance data mining on ‘normal’ data [29].

For a relatively recent overview of data mining and machine learning from Linked Data and the Semantic Web, see [30].

3. Graph Kernels for RDF

The Resource Description Framework (RDF)\(^1\) is the foundation of Linked Data and the Semantic Web. The central idea is to store statements about resources in subject-predicate-object form, called triples, which define relations between a set of terms. A triple \((s, p, o)\) in a set of triples \(\mathcal{T}\) specifies that the subject term \(s\) is related to the object term \(o\) by the predicate \(p\).

Often a set of triples \(\mathcal{T}\) is referred to and visualized as a graph, where the subject and object terms in the data constitute the vertices, and a triple \((s, p, o)\) specifies that \(s\) and \(o\) are connected by an edge with label \(p\). This graph interpretation breaks down when predicate terms are also used in the subject or

---

\(^1\)http://www.w3.org/standards/semanticweb/
object position, for example to define properties of the relations themselves [31]. However, most datasets that occur in practice define only a limited number of facts about the relations, so for simplicity we choose to interpret predicates in the predicate position and predicates in the subject or object positions as distinct, even if they have the same label.

More expressive Semantic Web knowledge representation formalisms, such as the Web Ontology Language (OWL) and RDF Schema (RDFS) can be represented in RDF. Therefore, RDF triple-stores, often include reasoning engines to automatically derive new triples, if the RDF represents data modeled using these more expressive formalisms.

Figure 1 is an illustration of a simple RDF graph of two Persons (person1 and person2) that authored a couple of papers, one of them (paper3) together.

Learning from RDF using graph kernels takes the approach that instances, which are vertices in a large RDF graph, are represented by their neighborhood, i.e. the triples around them, up to a certain depth. For example, in Figure 1, the instances could be person1 and person2, and their neighborhood of depth 1 would be all the triples in which they are the subject.

The graph in Figure 1 is an intuitive visualization of RDF data as a directed labeled multigraph. But when described formally, labeled multigraphs are more complicated than necessary for our purpose. Moreover, such a representation requires us to always distinguish between edges and vertices, which turns out not to be necessary in the description of any of our algorithms. So instead we opt to reify the edges: we represent the set of triples as vertices which allows the RDF to be represented by a simple directed graph with vertex labels.

In the following we use RDF graphs \( G = (V, E, \ell) \) based on sets of triples \( T \). For such a graph \( G \), the vertices \( V \) are defined as:

\[
V = \{ v \mid \exists x, y ((v, x, y) \in T \lor (x, y, v) \in T) \} \cup T. \tag{1}
\]

So there is a vertex for each unique subject and object, and for each triple. The edges \( E \) are:

\[
E = \{ (v, (v, p, o)) \mid (v, p, o) \in T \} \cup \{ ((s, p, v), v) \mid (s, p, v) \in T \}. \tag{2}
\]
So for each triple, two edges are created. Finally, we define the label function $\ell$ which assigns a label to a vertex in $V$ such that $\ell(v) = v$ if $v$ is a subject/object, and $\ell((s, p, o)) = p$ for vertices that represent triples. Figure 2 illustrates this representation by visualizing the graph in Figure 1 following this bipartite definition.

Figure 2: Small example RDF graph of an author network as a bipartite representation without edge labels.

This definition has the additional advantage that our algorithms, as well as the subsequent discussion, apply to any labeled simple graphs, regardless of whether or not they are constructed from a triple store.

3.1. Kernel Definitions

Below we define a family of graph kernels for RDF that are based on the idea of counting common substructures in the neighborhoods of the instances. The majority of the work on graph kernels for RDF fits into this family.

3.1.1. Convolution Kernel

The definition of the kernel is based on an extraction function $J$, which maps an instance vertex $i$ in graph $G$ to a set of graph substructures derived from $G$. These graph substructures are themselves graphs of the same form. The kernel is based on counting common, i.e. equivalent, substructures. Two substructures $G_1$ and $G_2$ are equivalent, denoted $G_1 \equiv G_2$, if their graphs are equivalent according to the usual definition of graph equivalence, with the additional requirement that the corresponding vertices have the same label. Thus, the equivalence class is always completely determined by the label function.

All the kernels that we define in the rest of this paper are instances of the following convolution kernel [32]. Let $G = (V, E, \ell)$ be an RDF graph and let $i, i' \in V$ be two instance vertices, then

$$k_J(i, i') = \sum_{x_i \in J(G, i)} \sum_{x_{i'} \in J(G, i')} 1[x_i \equiv x_{i'}]$$

where $1[x_i \equiv x_{i'}]$ is 1 if $x_i \equiv x_{i'}$ and 0 otherwise.
Different variants of the extraction function $J$ lead to different graph kernels for RDF graphs. In most of the kernels that we define the function $J$ is composed of two functions $H$ and $F$, where $H$ extracts the relevant neighborhood for an instance $i$ and $F$ extracts the substructures from that neighborhood. In this case $J_{H,F}(G, i) = F(H(G, i))$.

### 3.1.2. Instance Neighborhood

For each instance we consider either the tree or the graph around the instance $i$ extracted from the RDF graph $G$, up to depth $d$, as its neighborhood.

Let $\text{Graph}^d(G, i)$ extract the subgraph $G' = (V', E', \ell')$ around an instance vertex $i$ from graph $G$ up to depth $d$. The vertices $V'$ in this graph contain all vertices that are reachable in $G$ from $i$ in at most $d$ steps, and the same is true for the edges $E'$. The label function $\ell'$ is the same as $\ell$, but restricted to the vertices in $V'$. Note that when $d = 0$ we only extract the vertex $i$ (and no edges).

The function $\text{Tree}^d(G, i)$ extracts the subtree of depth $d$ rooted at instance node $i$ from a graph $G = (V, E, \ell)$. Define a walk of length $d$ from vertex $v_1 \in V$ as a sequence $v_1, v_2, \ldots, v_d$ of vertices such that for each $i < d$ there is an edge $(v_i, v_{i+1}) \in E$. Then $\text{Tree}^d(G, i) = (V', E', \ell')$, where the new vertex set $V'$ consists of all walks of length $d$ or less from vertex $i$ in $G$; two walks $v', w' \in V'$ are joined by an edge in $E'$ if and only if $w'$ extends $v'$ by one additional vertex. Finally the label function $\ell'$ maps a vertex $v' \in V'$ to the original label of the last vertex of the walk, that is, $\ell'(v_1, \ldots, v_n) = \ell(v_n)$. The root of $\text{Tree}^d(G, i)$ is the vertex sequence with just $i$, which is identified by its lack of incoming edges. In other words, $\text{Tree}$ constructs the tree up to depth $d$ by following the walks from the instance vertex $i$.

### 3.1.3. Substructures

Given a graph $G = (V, E, \ell)$ we consider different substructures to count, creating different kernels. Below we define 3 different types of substructures.

**Bag of Labels.** The most basic approach is to count the equivalent vertices in the neighborhood of the instance. Effectively this boils down to counting vertices with the same label, hence the name Bag of Labels (BoL). For a graph $G = (V, E, \ell)$, we extract the following substructures

$$\text{BoL}((V, E, \ell)) = \{(\{v\}, \emptyset, \ell) \mid v \in V\}.$$  \hspace{1cm} (4)

Thus, each substructure in the bag of labels is a very simple graph with one vertex and no edges.

**Walks.** To be able to define the extraction function we first need to extract all walks of a fixed length $n$ starting from a vertex $v_1$. For a graph $G = (V, E, \ell)$ and a vertex $v_1 \in V$, let

$$\text{Walks}^n((V, E, \ell), v_1) = \{\{(v_1, v_{i+1}) \mid i < n\}, \ell) \mid v_1, \ldots, v_n \text{ is a walk in } (V, E, \ell)\}.$$  \hspace{1cm} (5)
The extraction function then extracts all walks from the neighborhood \( G = (V, E, \ell) \) up to a given depth \( n \):

\[
Walks^{\leq n}((V, E, \ell)) = \bigcup_{0 \leq k \leq n} \bigcup_{v \in V} \text{Walks}^k((V, E, \ell), v) .
\] (6)

In [14] paths, i.e. walks that do not contain repeating vertices, are also considered. However, both that paper and [9] find very little difference in predictive performance between the two settings. Therefore, we stick to walks for clarity. Furthermore, in the case of instance trees, paths and walks are the same.

**Subtrees.** Finally, we consider full subtrees up to depth \( n \) of a graph \( G \). For this definition we use the \( \text{Tree} \) function defined earlier.

For a graph \( G = (V, E, \ell) \) the full subtrees up to depth \( n \) is the set given by:

\[
\text{Subtrees}^{\leq n}((V, E, \ell)) = \bigcup_{0 \leq k \leq n} \{ \text{Tree}^k((V, E, \ell), v) \mid v \in V \} .
\] (7)

Potentially, other, more complicated, substructures can also be considered, like subgraphs and partial subtrees, but they can lead to significantly longer computation time and/or combinatorial explosion.

**Root Constraint.** For trees \( G \) we also consider the constraint in which only substructures are counted that start from the root vertex, because it has been used before [14, 10] and it can lead to faster computation. The \( \text{Root} \) version of an extraction function \( F \) returns only those substructures that start with the root vertex:

\[
F_{\text{Root}}(G) = \{ G' \in F(G) \mid \text{the root of } G \text{ is the root of } G' \} .
\] (8)

In the bag of labels case, this would make little sense, since this would lead to only one label. But in the \( \text{Walks} \) setting this would mean that all walks start in the root vertex and in the \( \text{Subtrees} \) setting that all subtrees have the root vertex as their root vertex.

**3.1.4. Possible Kernels**

These different options can be combined into 8 possible kernels. For example, given an RDF graph \( G \), the combination of instance trees with counting walks is:

\[
k_{\text{Tree}^d, \text{Walks}^{\leq n}}(i, i') = \sum_{x_i \in \text{Walks}^{\leq n}(\text{Tree}^d(G, i))} \sum_{x_i' \in \text{Walks}^{\leq n}(\text{Tree}^d(G, i'))} 1[x_i \equiv x_i'] .
\] (9)
The full list of kernels is the following.

\[
\begin{align*}
\text{GraphBoL}^d &= k^d_{\text{Graph}, \text{BoL}} \quad (10) \\
\text{TreeBoL}^d &= k^d_{\text{Tree}, \text{BoL}} \quad (11) \\
\text{GraphWalks}^{d,n} &= k^d_{\text{Graph}, \text{Walks} \leq n} \quad (12) \\
\text{TreeWalks}^{d,n} &= k^d_{\text{Tree}, \text{Walks} \leq n} \quad (13) \\
\text{TreeWalksRoot}^{d,n} &= k^d_{\text{Tree}, \text{Walks} \leq n, \text{Root}} \quad (14) \\
\text{GraphSubtrees}^{d,n} &= k^d_{\text{Graph}, \text{Subtrees} \leq n} \quad (15) \\
\text{TreeSubtrees}^{d,n} &= k^d_{\text{Tree}, \text{Subtrees} \leq n} \quad (16) \\
\text{TreeSubtreesRoot}^{d,n} &= k^d_{\text{Tree}, \text{Subtrees} \leq n, \text{Root}} \quad (17)
\end{align*}
\]

\text{Graph} indicates that instance graphs are extracted from RDF and \text{Trees} that we use trees. \text{BoL} indicates the bag of labels substructures, \text{Walks} the walks substructures and \text{Subtrees} the subtrees. Finally \text{Root} indicates whether we use the root constraint. All the kernels have the depth parameter \(d\) which indicates to which depth the graphs or trees have been extracted. Kernels 3-8 also have a parameter \(n\) for the multiset of substructures, which is either the maximum walk length or the maximum subtree depth.

\text{Relation to Existing Kernels for RDF.} The TreeWalksRoot kernel is the same as the intersection path kernel defined in [10], which is very similar to the intersection subtree kernels from [14] and has almost identical performance. Both kernels consider only substructures that contain the root vertex. However, the original intersection subtree kernels in [14] count (partial) subtrees in the intersection tree, which are more difficult and computationally expensive to enumerate and hence the kernel does not easily fit this framework.

The GraphWalks kernel is nearly identical to the intersection graph walk kernel from [14]. Both kernels count common walks between two instances. The intersection graph walk kernel first creates the intersection graph between two instances and then counts the walks in that graph, whereas the GraphWalks kernel enumerates the walks in the instance graphs directly and then counts how many there are in common.

The GraphSubtrees kernel is actually the Weisfeiler-Lehman kernel [8] implemented for RDF. However, there is a difference with respect to repeatedly counting the same subtrees, which we will deal with in the following section on the computation of the kernels.

In [16] the GraphBoL kernel is used in the context of a recommender system. Multiple more complicated weighting schemes for the bag of labels in the neighborhood of vertex are tried in that paper but the variant that is essentially the GraphBoL kernel gives the best performance. The neighborhood-based graph kernel in [15] is also a GraphBoL kernel, but with a special weighting scheme taking into account the frequency and distance to the instance vertex of a label.
So, previous graph kernel research for RDF fits the framework, presented above, very well. Furthermore, this framework also introduces some new variants.

3.2. Kernel Computation

All of the kernels above can be computed using a map: \( f : \mathcal{X} \rightarrow \mathbb{N} \), from the set of all substructures \( \mathcal{X} = \bigcup_i J(G, i) \) to indices in a feature vector. With this map we can construct a feature vector \( \phi_i \) for each instance \( i \), which counts the occurrence of specific substructures for \( J(G, i) \). Then the convolution kernel defined earlier, is simply the dot-product between two feature vectors:

\[
k(i, i') = \phi_i \cdot \phi_{i'}.
\] (18)

The advantage of the feature vector representation is that it leads to efficient computations of the kernels [33] and for larger datasets the feature vectors can be used directly as input to linear learning methods, such as linear Support Vector Machines [34]. The sparse nature of these feature vectors fits very well with these linear Support Vector Machines.

The \textit{Subtrees} kernels can be computed using the Weisfeiler-Lehman algorithm [8]. This algorithm is efficient because only full subtrees are considered, since this greatly restricts the number of possible substructures. However, this leaves open the possibility of modifying the label function and thereby the equivalence relation, which we consider in Section 3.4.1.

The walks counted in the \textit{Walks} kernel can be straightforwardly enumerated for the instance graphs and trees using a recursive function. However, we can do this a bit more efficiently using an algorithm similar to the Weisfeiler-Lehman algorithm. The bag of labels kernels are special (base) cases of either of these algorithms.

3.2.1. Counting Subtrees Using Weisfeiler-Lehman

The Weisfeiler-Lehman Subtree graph kernel [8], from now on the Weisfeiler-Lehman (WL) kernel, is a state-of-the-art, efficient kernel for graph comparison. The kernel computes the number of subtrees shared between two (or more) graphs by using the Weisfeiler-Lehman test of graph isomorphism. This algorithm creates labels representing subtrees in a number of iterations. However, labels created in different iterations are different, but can potentially identify the same subtree. To make sure that a subtree is not counted more often than it occurs, the WL algorithm needs to be modified.

The rewriting procedure of Weisfeiler-Lehman kernel for RDF graphs (or trees) \( G \) is given in Algorithm 3.1, which is adapted from [8]. First, the algorithm creates a multiset label for each vertex based on the labels of the neighbors of that vertex (steps 1a and 1b). This multiset is sorted and together with the original label concatenated into a string, which is the new label (steps 2a and 2b). For each unique string a new (shorter) label is introduced and this replaces the original vertex label (steps 3 and 4). Add the end of each iteration, each
Algorithm 3.1 differs from the original algorithm in two places. First, our graphs have directed edges, which is reflected in the fact the neighborhood \( N(v) \) of a vertex \( v \) contains only the vertices reachable via outgoing edges. Secondly, as mentioned, in the original algorithm, labels from two iterations can potentially be different while still representing the same subtree. To make sure that this does not happen we have added tracking of the neighboring labels in the previous iteration, via the multiset \( M'(v) \), to the steps 1a and 1b. When the multiset of the current iteration is identical to that of the previous iteration, we simply give a vertex the label created in the previous iteration, instead of a new one, see step 2b.

Using the rewriting techniques of Algorithm 3.1 we can define the Weisfeiler-Lehman (WL) kernel. Let \( G_{i,n} = (V,E, l_n, l_{n-1}) \) and \( G'_{i',n} = (V',E', l_n, l_{n-1}) \) be the \( n \)-th iteration rewriting of the graphs (or trees) \( G_i \) and \( G'_{i'} \), for instances \( i, i' \), using Algorithm 3.1, and \( h \) the number of iterations. Note that we added the labels from the previous iteration to the graphs (and assume these to be empty when \( n = 0 \)). Then the Weisfeiler-Lehman kernel is defined as:

\[
k_W^h(i, i') = \sum_{n=0}^{h} k_{\delta, WL}(G_{i,n}, G'_{i',n}) ,
\]

where

\[
k_{\delta, WL}((V,E,l,l'), (V',E',l,l')) = \sum_{v \in V} \sum_{v' \in V'} [\delta(l(v), l(v')) \cdot (1 - \delta(l(v), l'(v))) \cdot (1 - \delta(l(v'), l'(v')))] .
\]

and \( \delta \) is the Dirac-kernel which tests for equality. This kernel counts the common vertex labels in each of the iterations of the graph rewriting process. Every label \( l_n(v) \) represents a subtree rooted in \( v \). By multiplying with the terms \( (1 - \delta(l(v), l'(v))) \) and \( (1 - \delta(l'(v), l'(v'))) \) we make sure that a label is only counted when it is not equal to the label for that vertex in the previous iteration. We can do this because labels for equivalent subtrees are constructed in the same iteration. This addition ensures that the same subtree is not counted multiple times, which is possible in the original Weisfeiler-Lehman definition. So the kernel \( k_W \) essentially defines the Subtrees kernels: \( \text{GraphSubtrees}^{d,n} \), \( \text{TreeSubtrees}^{d,n} \) or \( \text{TreeSubtreesRoot}^{d,n} \), depending on whether we used graphs or trees as input and whether we restrict ourselves to only the root vertex.

The label dictionary \( f \) can be used as our mapping function\(^2\) to construct feature vectors \( \phi \) to compute the kernels by taking the dot-product, or to be used directly. When \( h = 0 \) we get a bag of labels kernel.

\(^2\)For instance when we use consecutive integers as compressed labels.
Algorithm 3.1 Weisfeiler-Lehman Relabeling

**Input** a set of graphs \( G \) and number of iterations \( h \)

**Output** (shared) label functions \( l_0 \) to \( l_h \) and label dictionary \( f \)

**Comments** \( M_n(v) \) are multisets of labels for a vertex \( v \). \( N(v) \) is the neighborhood of \( v \), which is \( N(v) = \{(v,v') \in E\} \). \( M'_n(v) \) are multisets of labels for a vertex \( v \) which contain the previous iteration.

- for \( n = 0 \) to \( h \)
  - for each \( (V,E,\ell) \in G \)
    1. Multiset-label determination
      - for each \( v \in V \)
        a. if \( n = 0 \), \( M_n(v) = \ell_0(v) = \ell(v) \) and \( M'_n(v) = \emptyset \)
        b. if \( n > 0 \), \( M'_n(v) = M_n(v) \) and \( M_n(v) = \{l_{n-1}(u)|u \in N(v)\} \)
    2. Sorting each multiset
      a. for each \( M_n(v) \), sort the elements in \( M_n(v) \) in ascending order and concatenate them into a string \( s_n(v) \)
      b. for each \( s_n(v) \), if \( n > 0 \) and \( M'_n(v) \) is unequal to \( M_n(v) \), add \( l_{n-1}(v) \) as a prefix to \( s_n(v) \), else if \( n > 0 \), set \( s_n(v) = s_{n-1}(v) \)
    3. Label compression
      - for each \( s_n(v) \), map \( s_n(v) \) to a (new) compressed label, using a function \( f : \Sigma^* \rightarrow \Sigma \), such that \( f(s_n(v)) = f(s_n(v')) \) iff \( s_n(v) = s_n(v') \)
    4. Relabeling
      - for each \( s_n(v) \), set \( l_n(v) = f(s_n(v)) \)
3.2.2. Counting Walks

In the vein of the Weisfeiler-Lehman algorithm we can also count the walks, in the instance graphs or trees, in iterations. This algorithm is given in Algorithm 3.2. In each iteration \( n \) the walks of length \( n \) are determined. This is achieved by first creating a union of all walk labels from the neighbors of \( v \) (steps 1a and 1b). Then all these labels are prefixed with the original vertex label \( \ell(v) \) (step 2). Each of these concatenated labels is mapped and replaced with a unique new label (step 3 and 4), as in the Weisfeiler-Lehman algorithm. Thus at the end of each iteration, each vertex contains a set of labels that represent the walks of length \( n \) starting from that vertex. Note that there is no label sorting step in this algorithm and that the label functions \( l_n \) map to sets of labels.

**Algorithm 3.2 Walk Count**

**Input** a set of graphs \( \mathcal{G} \) and number of iterations \( h \)

**Output** (shared) label functions \( l_0 \) to \( l_h \), which map a vertex \( v \) to a set of labels, and label dictionary \( f \)

- for \( n = 0 \) to \( h \)
  - for each \( (V, E, \ell) \in \mathcal{G} \)
    1. Multiset-label determination
      - for each \( v \in V \)
        a. if \( n = 0 \) set \( l_0(v) = \{\ell(v)\} \)
        b. if \( n > 0 \) set \( l_n(v) = \bigcup_{u \in N(v)} l_{n-1}(u) \)
    2. Concatenation
      - for each \( \lambda_v \in l_n(v) \), replace \( \lambda_v \) with the concatenation of \( \lambda_v \) and \( \ell(v) \)
    3. Label compression
      - for each \( \lambda_v \in l_n(v) \), map \( \lambda_v \) to a new compressed label, using a function \( f : \Sigma^* \to \Sigma \), such that \( f(\lambda_v) = f(\lambda_{v'}) \) iff \( \lambda_v = \lambda_{v'} \)
    4. Relabeling
      - for each \( \lambda_v \in l_n(v) \), replace \( \lambda_v \) with \( f(\lambda_v) \)

Using Algorithm 3.2 we define the Walk Count (WC) kernel. Let \( G_{i,n} = (V,E,l_n) \) and \( G'_{i',n} = (V',E',l_n) \) be the \( n \)-th iteration rewriting of the graphs (or trees) \( G_i \) and \( G_{i'} \), for instances \( i, i' \), using Algorithm 3.2, and \( h \) the number
of iterations. Then:

\[ k_{\text{WC}}^h(i, i') = \sum_{n=0}^{h} k_{\delta, \text{WC}}(G_i, G_{i'}) \]  \hspace{1cm} (21)

where

\[ k_{\delta, \text{WC}}((V_i, E_i, l_n), (V_{i'}, E_{i'}, l_n)) = \sum_{\lambda_v \in l_n(v), v \in V_i} \sum_{\lambda_{v'} \in l_n(v'), v' \in V_{i'}} \delta(\lambda_v, \lambda_{v'}) \]  \hspace{1cm} (22)

In this kernel every label \( \lambda \) represents a different walk. Similar to the WL kernel, this kernel defines the Walks kernels: \( \text{GraphWalks}^{d,n} \), \( \text{TreeWalks}^{d,n} \) or \( \text{TreeWalksRoot}^{d,n} \), depending on whether we used graphs or trees as input and whether we restrict ourselves to only the root vertex. Again, we can use the label function \( f \) to create feature vectors and when \( h = 0 \) we have the bag of labels kernel.

### 3.3. Direct Graph Kernels

As mentioned, the intuition of graph kernels for RDF is that the neighborhood of an instance vertex contains the interesting information about that instance in the form of substructures. Which substructures are used is determined by the function \( J \). Up until now we have defined this function in two stages, by using a function \( H \) to extract the neighborhood and \( F \) to compute the features in this neighborhood. However, it is also possible to define \( J \) without using an extraction function \( H \), but directly on the graph \( G \). To do so we need to determine which substructures are sufficiently close to the instance vertex to be included as features for that instance.

#### 3.3.1. Kernel Definitions

Let \( \text{dist}(v, v') \) define the length of the shortest path in \( G \) between the vertices \( v \) and \( v' \), and let \( G = (V, E, \ell) \) be an RDF graph. Furthermore, let the function \( F^n(G, v) \) return a set of substructures of size \( n \in \mathbb{N} \), rooted in the vertex \( v \). Then we define a direct extraction function \( J \) as follows.

\[ \text{Direct}_F^d((V, E, \ell), i) = \{ x \in F^n((V, E, \ell), v) \mid \text{dist}(i, v) + n \leq d, v \in V \} \]  \hspace{1cm} (23)

Thus, for an instance vertex \( i \) in \( G \) this function returns all the substructures that are sufficiently close to \( i \). Close is defined as the sum of the distance between the root vertex of the structure and the instance vertex and the size of the substructure. So, larger substructures have to have a root vertex closer to the instance vertex. This definition ensures that no vertex \( v' \) in a substructure has \( \text{dist}(i, v') > d \), since \( \text{dist}(i, v) + \text{dist}(v, v') \geq \text{dist}(i, v') \) and \( n \geq \text{dist}(v, v') \).

For the function \( F \) we have two obvious candidates, the \( \text{Walks}^n \) and \( \text{Tree}^d \) functions defined earlier. Note that the \( \text{Tree} \) function returns a graph and
not a set of graphs. We simply wrap this function as follows: \( \text{Trees}^d(G,v) = \{ \text{Trees}^d(G,v) \} \). This leads to the following two additional kernels.

\[
\begin{align*}
\text{DirectWalks}^d & = k_{\text{DirectWalks}}^d, \\
\text{DirectSubtrees}^d & = k_{\text{DirectSubtrees}}^d.
\end{align*}
\]

These two kernels are variants of the \textit{Walks} and \textit{Subtrees} kernels defined earlier, but are directly defined on the RDF graph \( G \), instead of using subgraphs or subtrees. A \textit{DirectBoL} variant is also possible, however, it is easy to see that this is equivalent to the \textit{GraphBoL} kernel. Both kernels have only one parameter \( d \) instead of the two parameters \( d \) and \( n \) that we saw earlier.

3.3.2. Kernel Computation

Both the \textit{Direct} kernels defined above can be computed using the algorithms that we have defined earlier.

Let \( G_n = (V,E,l_n,l_{n-1}) \) be the \( n \)th-iteration of one RDF graph rewritten for \( h \) iterations using Algorithm 3.1, with \( l_0 \) to \( l_h \) the resulting label functions. Note that we again add the previous iteration of the label function to the graph. Then we compute the Direct Weisfeiler-Lehman kernel between two instances \( i,i' \in I \), as:

\[
k_{\text{WLDirect}}^{d,h}(i,i') = \sum_{n=0}^{h} k_{\text{WLDirect}}^{d,n}( (G_n,i), (G_n,i') ).
\]

where

\[
k_{\text{WLDirect}}^{d,h}( ( (V,E,l_n,l_{n-1}),i ), ( (V,E,l_n,l_{n-1}),i' ) ) = \sum_{\text{dist}(i,v)+h \leq d,v \in V} \sum_{\text{dist}(i',v')+h \leq d,v' \in V} [ \delta(l(v),l(v')) \cdot (1 - \delta(l(v),l'(v))) \cdot (1 - \delta(l(v'),l'(v')))] .
\]

This definition is very similar to the earlier definition of the Weisfeiler-Lehman kernel. The main difference is that the neighborhood of an instance is now defined by the distance function \textit{dist}. The \textit{WLDirect} kernel implements the \textit{DirectSubtrees} kernel.

This kernel is similar to the fast approximation of the Weisfeiler-Lehman kernel for RDF introduced in [9]. The main difference is that in the definition in the current paper, repeated counts of the same subtree are not allowed. Two other differences are that the direction that the labels ‘follow’ during the relabeling is reversed and that the iterations of the WL algorithm are weighted. If we modify the algorithm in [9] to incorporate these differences, then it will result in the same kernel.

Let \( G_n = (V,E,l_n) \) be the \( n \)th-iteration of one RDF graph rewritten for \( h \) iterations using Algorithm 3.2, and \( l_0 \) to \( l_h \) the resulting label functions. Then we compute the Direct Walk Count kernel between two instances \( i,i' \in I \), as:
\[ k_{WCDirect}^{d,h}(i, i') = \sum_{n=0}^{h} k_{\delta,WCDirect}^{d,n}((G_n, i), (G_n, i')), \tag{28} \]

where

\[ k_{\delta,WCDirect}^{d,h}(((V,E,l_n), i), ((V,E,l_n), i')) = \sum_{\lambda_v \in l_n(v), \text{dist}(i,v) + h \leq d} \delta(\lambda_v, \lambda_{v'}). \tag{29} \]

Again, the main difference with the earlier Walk Count kernels is that the neighborhood of an instance is defined using the distance function \( \text{dist} \). The WCDirect kernel implements the DirectWalks kernel.

For the computation of both kernels, the RDF graph \( G \) can be restricted to those vertices that are actually within a distance \( d \) of at least one instance vertex \( i \). Furthermore, in the for loop over the vertices \( V \) in Algorithms 3.1 and 3.2, we can exclude vertices \( v \) for which \( \neg \exists i \in I : \text{dist}(i, v) + n \leq d \). In other words, we can ignore vertices that are too far from all instance vertices for the current iteration \( n \). Also, tracking which vertices are in the neighborhood of an instance \( i \), i.e. within \( \text{dist} \leq d \), can be done by creating a map from vertices to integers (indicating the depth at which the vertex occurs) for each instance.

**Computational Complexity.** In [8] it is shown that the runtime for the Weisfeiler-Lehman relabeling algorithm on a set of graphs is \( O(Nhm) \), where \( N \) is the number of graphs, \( h \) is the number of iterations and \( m \) is the number of vertices per graph. For our Direct kernels, we do not have \( N \) graphs, but just a single larger graph. Our larger graph has a number of vertices \( k \). So for these kernels the relabeling algorithm has a runtime complexity \( O(hk) \). Hence, in situations with \( k < Nm \), the Direct kernels will be faster. This scenario is typical for the RDF use-case, where the subgraphs for each instance share a (large) number of vertices and edges, which means \( Nm \gg k \) given large enough \( N \).

For the Walk Count kernel the situation is identical, as can be seen from the large similarity with the Weisfeiler-Lehman algorithm.

**Fast Tree Kernels.** To compute the Tree kernels defined earlier it is not necessary to explicitly extract each tree from the RDF graph, it is sufficient to retain a ‘tree-view’ for each instance and use the same trick with the distance function as in the definition of the two Direct kernels. This means that we only need to perform the Weisfeiler-Lehman/Walk Count once on the RDF graph \( G \) instead of on all the extracted subtrees, which is significantly faster. Therefore, in the experiments in this paper we use such an implementation.

### 3.4. Dealing with Unique Labels

Two of the main differences RDF graphs and the typical graphs used in graph mining and machine learning from graphs, are that vertices in RDF graphs have
unique labels\textsuperscript{3} and there is a large number of different labels overall. This leads to very specific graph patterns, which do not generalize well. In this section we present two strategies to deal with these problems.

3.4.1. Removing Low Frequency Labels

Ideally, a form of inexact matching between the too specific graph patterns could help alleviate this problem. However, this is potentially very computationally expensive. As is shown in [33] we can remain efficient as long as we consider equivalence classes of the substructures. In this section we introduce adaptations of the kernel computation algorithms that allow for some inexact matching, by changing the equivalence classes.

For clarity, our adaptations are defined directly on the algorithms that compute the kernels (Weisfeiler-Lehman and Walk count). They are based on the idea that labels that occur only in the neighborhood of an instance for a low number of instances are not very informative. The frequency of a label $\lambda$ in a set of graphs (or trees) $\mathcal{G}$ is straightforwardly defined as follows:

$$Freq(\lambda, \mathcal{G}) = |\{(V, E, \ell) \in \mathcal{G} | \exists v \in V : \ell(v) = \lambda\}|. \quad (30)$$

$Freq$ simply determines the number of graphs that $\lambda$ occurs in at least once. Note that $\ell(v)$ can be the original label function $\ell$ or the variants $l_0, \ldots, l_h$ created by iterations of the Weisfeiler-Lehman algorithm.

We also define the frequency of a label $\lambda$ when we use Direct kernels as follows, let $(V, E, \ell)$ be an RDF graph, $\mathcal{I}$ a set of instance vertices, and $r$ a distance constraint, then

$$Freq(\lambda, (V, E, \ell), \mathcal{I}, r) = |\{i \in \mathcal{I} | \exists v \in V : \ell(v) = \lambda, dist(i, v) \leq r\}|, \quad (31)$$

where $r = d - n$, with $d$ the size of the neighborhood and $n$ the current iteration of the Weisfeiler-Lehman algorithm.

We then modify the Weisfeiler-Lehman algorithm in two ways. These two modifications are given in Algorithm 3.3. For clarity, we only give the lines that are changed from the original algorithm. The first modification is that the sets of labels $M_n(v)$ (and $M'_n(b)$) are now sets instead of multisets, so multiple neighbors that have the same label are reduced to one label. The second modification is to only use a label in the rewriting when it is above a minimal frequency threshold $\theta$, which is shown in the new lines 1b and 2b. For the DirectSubtrees kernel we use the second definition of $Freq$.

These two changes to the Weisfeiler-Lehman algorithm allow for inexact matching of subtrees because we only look at labels (and therefore subtrees) that are frequent enough and we also ignore multiple instances of the same label in the neighborhood of vertex.

For the Walk Count algorithm we need two definitions for the frequency of a walk. For a set of graphs $\mathcal{G}$ the frequency of a walk label $\lambda$ is defined as:

$$Freq_{Walk}(\lambda, \mathcal{G}) = |\{(V, E, \ell) \in \mathcal{G} | \exists v \in V : \lambda \in \ell(v)\}|. \quad (32)$$

\textsuperscript{3}In our bipartite representation this holds for the vertices based on the subject and objects.
Algorithm 3.3 Weisfeiler-Lehman Relabeling Modification

**Input** Minimum frequency threshold $\theta$

**Comments** $M_n(v)$ are sets of labels for a vertex $v$.

1. Multiset-label determination
   - for each $v \in V$
     - if $n > 0$, $M'_n(v) = M_n(v)$ and $M_n(v) = \{l_{n-1}(u) | u \in N(v), \text{Freq}(l_{n-1}(u), (V, E, l_{n-1})) \geq \theta\}$

2. Sorting each multiset
   - for each $s_n(v)$, if $n > 0$ and $M'_n(v)$ is unequal to $M_n(v)$ and $\text{Freq}(l_{n-1}(v), (V, E, l_{n-1})) \geq \theta$, add $l_{n-1}(v)$ as a prefix to $s_n(v)$, else if $n > 0$, set $s_n(v) = s_{n-1}(v)$

This definition is analogous to the definition of the frequency of a label for the Weisfeiler-Lehman algorithm above.

A similar definition follows for the DirectWalks kernel. Let $(V, E, l)$ be an RDF graph, $I$ a set of instance vertices, and $r = d - n$ a distance constraint, then the frequency of a walk label $\lambda$ is:

$$\text{Freq}_{\text{Walk}}(\lambda, (V, E, l), I, r) = |\{i \in I | \exists v \in V : \lambda \in l(v), \text{dist}(i, v) \leq r\}|.$$  \hspace{1cm} (33)

Algorithm 3.4 gives the modification to the Walk Count algorithm. The change to use sets instead of multisets is not applicable to the Walk Count algorithm. The modification to use the minimal frequency threshold $\theta$ is similar to the Weisfeiler-Lehman algorithm. In line 1b we only include the walk $\lambda$ if its frequency is high enough and in line 2b we only use the label $l(v)$ if its frequency is high enough. For the DirectWalks kernel we use the second definition of $\text{Freq}_{\text{Walk}}$.

Algorithm 3.4 Walk Count Modification

**Input** Minimum frequency threshold $\theta$

1. Multiset-label determination
   - for each $v \in V$
     - if $n > 0$ set $l_n(v) = \{\lambda \in l_{n-1}(u) | \text{Freq}_{\text{Walk}}(\lambda, (V, E, l_{n-1})) \geq \theta, u \in N(v)\}$

2. Concatenation
   - if $\text{Freq}(l(v), (V, E, l_{n-1})) \geq \theta$, for each $\lambda_v \in l_n(v)$, replace $\lambda_v$ with the concatenation of $\lambda_v$ and $l(v)$
Essentially, the changes to the Weisfeiler-Lehman and Walk count algorithms changes the equivalence classes induced by the equivalence relation $\equiv$ used in Equation 3. For example, suppose we have the following two walks (as sequences of labels): $a, b, c$ and $b, a, c$. If the frequency of the label $b$ is below $\theta$, then these walks are now considered equal.

We will refer to the 8 new kernels (i.e. each variant of the Walks and Subtrees kernel), introduced above, as $\text{MinFreq}$ kernels.

### 3.4.2. Hub Removal

Another direction for tackling the specific label problem is to replace these labels with more generic labels. In [11] a number of hub removal strategies were studied to perform this task. The most promising approach detected hubs based on the combination of edge and vertex labels and relabeled connected vertices with the label of the hub vertex. In this section we introduce a slightly more generic version of this approach.

Algorithm 3.5 presents our hub removal strategy.\footnote{Note that we call it hub removal because of historical reasons, but the ‘hubs’ that are removed are actually edges (or in our bipartite case the edges that represent predicates), not vertices.} First, all subject-predicate and predicate-object pairs in a set of RDF triples are counted and sorted by frequency. All pairs with a minimum frequency $k$ are kept. We furthermore remove the pairs that have an instance as either subject or object, because we do not want to potentially remove instance vertices. The remaining pairs represent new more generic labels, which we want to put as new labels on the vertices. For the remaining pairs we create a map $\text{Freq}$ from pairs to their frequency. Then we loop over our RDF graph $G$ and for each vertex $v$ that represents a subject or object, we check its outgoing and incoming neighbors (with their connected vertex) for occurrences in our map $\text{Freq}$. If the pair occurs in $\text{Freq}$ then we remove it from the graph. The pair that has the lowest frequency provides the new label for $v$, by concatenating the labels of the two elements of the pair. The intuition behind using the pair with the lowest frequency is that we want a more generic label for the vertex, but we do not want it to be overly generic. So we take the label that is still considered frequent (since it is above $k$), but not the most frequent.

Compared to the algorithms in [11], this algorithm is more rigorous, since it removes all the frequent pairs from the graph. Furthermore, we use the parameter $k$ to control the edges to remove by specifying a minimum frequency, instead of removing the top $h$ number of hubs. This hub removal algorithm tries to tackle both the problem of a large number of different labels as well as the problem of unique vertex labels by replacing vertex labels by more generic labels.
Algorithm 3.5 Hub Removal

**Input** a set of triples $\mathcal{T}$, a set of instances $\mathcal{I}$, graph $G$ based on $\mathcal{T}$, and minimum frequency $k$

**Output** graph $G$ with hubs removed

**Comments** $N_{\text{out}}(v) = \{v' \mid (v, v') \in E\}$ is the out-neighborhood and $N_{\text{in}}(v) = \{v' \mid (v', v) \in E\}$ is the in-neighborhood

1. count the subject-predicate $(s, p)$ and predicate-object $(p, o)$ pairs in $\mathcal{T}$ and retain the pairs with a frequency $\geq k$
2. remove all pairs for which the subject or object is an instance in $\mathcal{I}$
3. for all retained pairs $\rho$, create a map $\text{Freq}(\rho)$ which maps $\rho$ to its frequency
   - for each $v \in V$ that represents subject or object
     - $c_{\text{min}} = \infty$
     - for each $v' \in N_{\text{out}}(v)$
       - $v''$ is the vertex for which $(v', v'') \in E$
       - if $\text{Freq}(\ell(v'), \ell(v''))$ is defined
         - Remove $v'$ from $V$ and the edges $(v, v')$ and $(v', v'')$ from $E$
         - if $\text{Freq}(\ell(v'), \ell(v'')) < c_{\text{min}}$
           - $c_{\text{min}} = \text{Freq}(\ell(v'), \ell(v''))$
           - $\rho_{\text{min}} = (\ell(v'), \ell(v''))$
     - for each $v' \in N_{\text{in}}(v)$
       - $v''$ is the vertex for which $(v'', v') \in E$
       - if $\text{Freq}(\ell(v''), \ell(v'))$ is defined
         - Remove $v'$ from $V$ and all edges $(v'', v')$ and $(v', v)$ from $E$
         - if $\text{Freq}(\ell(v''), \ell(v')) < c_{\text{min}}$
           - $c_{\text{min}} = \text{Freq}(\ell(v''), \ell(v'))$
           - $\rho_{\text{min}} = (\ell(v''), \ell(v'))$
    - if $c_{\text{min}} \neq \infty$, set $\ell(v)$ to the concatenation of $\rho_{\text{min}}$
   - Removed any orphaned $v \in V$
4. Experiments

In this section we present results for five sets of experiments using the kernels presented above. The first four sets are classification experiments with Support Vector Machines (SVMs): the first using the kernels on regular datasets, the second using the MinFreq kernels, the third using the kernels in combination with hub removal, and finally using the kernels on unlabeled RDF graphs. The final set of experiments presents the runtimes for the different kernels.

The goal of these experiments is to investigate the performance differences between the kernel options provided by the presented framework. In earlier experiments [9] we used iteration weighting in the Weisfeiler-Lehman kernel and class weights in the SVMs. Also the direction of the Weisfeiler-Lehman algorithm in [9] was opposite, i.e. labels ‘traveled’ to the leafs. In this scenario the WL kernel does not fit our framework. To have as few potential influencing factors as possible, we do not use these settings in the experiments in this paper. However, they can have a positive effect on performance, so they are options to consider when applying the kernels in a practical application.

Where appropriate, we also include the IntersectionSubTree and IntersectionPartialSubTree kernels from [14] for completeness, using settings that are provided in that paper. The GraphWalks kernels are so similar to the intersection graph kernels from [14], that we do not include the latter. Moreover, the intersection graph kernels are very inefficient to compute.

All of the kernels and experiments are implemented in Java and are available online\textsuperscript{5}. For our experiments we use the SESAME\textsuperscript{6} triple-store to handle RDF data and the Java versions of the LibSVM [35] and LibLINEAR\textsuperscript{7} [34] support vector machine libraries.

In each classification experiment we optimize over a number of extraction depths \(d\) for the graphs and trees. For the maximum walk length and maximum subtrees depth \(n\) (or equivalently the number of iterations parameter \(h\)) we use \(n = d\). We choose not to additionally optimize over \(n\) during training, because higher settings include the lower settings and thus the optimization has very little effect, other than substantially increasing training time. Furthermore, we test the kernels both in the setting where additional triples are added to the dataset by RDFS inferencing by the triple-store and when these are not. Since the differences in results between these two settings are not very large, we only report the results for the setting which gave the best performance. As done earlier in [14, 9], we set the labels for the instance vertices to one identical special root label, since the original label uniquely identifies each instance, which is not useful. All computed kernels/feature vectors are normalized before use.


\textsuperscript{6}http://www.openrdf.org/

\textsuperscript{7}http://liblinear.bwaldvogel.de/
4.1. Regular Classification

Table 1 presents the average number of edges per instance graph or tree for the datasets used in the classification experiments. For the Direct kernels, the size of the neighborhood that is considered is the same as for the instance graphs.

Table 1: Average number of edges for the extracted instance graphs and trees for the different datasets. The extraction depth is given by \( d \) and 'inf' indicates inferencing by the triple-store.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Graph ( d=2 )</th>
<th>Graph ( d=4 )</th>
<th>Graph ( d=6 )</th>
<th>Tree ( d=2 )</th>
<th>Tree ( d=4 )</th>
<th>Tree ( d=6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Affiliation</td>
<td>42</td>
<td>571</td>
<td>3135</td>
<td>42</td>
<td>571</td>
<td>15640</td>
</tr>
<tr>
<td>Affiliation, inf.</td>
<td>64</td>
<td>1261</td>
<td>5352</td>
<td>64</td>
<td>1261</td>
<td>24112</td>
</tr>
<tr>
<td>Mutag</td>
<td>137</td>
<td>438</td>
<td>506</td>
<td>137</td>
<td>438</td>
<td>929</td>
</tr>
<tr>
<td>Mutag, inf.</td>
<td>139</td>
<td>778</td>
<td>1080</td>
<td>139</td>
<td>778</td>
<td>3824</td>
</tr>
<tr>
<td>Lithogenesis</td>
<td>67</td>
<td>351</td>
<td>1090</td>
<td>67</td>
<td>351</td>
<td>1487</td>
</tr>
<tr>
<td>Lithogenesis, inf.</td>
<td>75</td>
<td>461</td>
<td>1387</td>
<td>75</td>
<td>461</td>
<td>2538</td>
</tr>
<tr>
<td>Theme</td>
<td>51</td>
<td>409</td>
<td>3245</td>
<td>51</td>
<td>409</td>
<td>3720</td>
</tr>
<tr>
<td>Theme, inf.</td>
<td>57</td>
<td>492</td>
<td>3766</td>
<td>57</td>
<td>492</td>
<td>4850</td>
</tr>
<tr>
<td>Category</td>
<td>50</td>
<td>722</td>
<td>-</td>
<td>50</td>
<td>722</td>
<td>-</td>
</tr>
<tr>
<td>Category, inf.</td>
<td>54</td>
<td>835</td>
<td>-</td>
<td>54</td>
<td>835</td>
<td>-</td>
</tr>
</tbody>
</table>

From the table we can see that the number of additional edges that the tree perspective introduces with respect to the graph perspective differs quite significantly between datasets. This difference is an indication of the tree/graph like nature of the underlying RDF graph. If this RDF graph contains a large number of cycles, then the instance trees will have far more edges than the instance graphs.

4.1.1. Affiliation Prediction

Our first classification experiment uses the AIFB dataset, used for the first time in [17] and repeated in following papers. This dataset contains information about the AIFB research institute: people, publications, groups; modeled in the SWRC ontology [36]. In total this dataset contains 178 persons that belong to 1 of 5 research groups. One of these groups has only 4 members, which we ignore. The goal of the experiment is to predict the affiliation for the remaining 174 persons. For training and testing the affiliation relation (and the inverse employs relation) are removed from the dataset.

For each kernel a C-SVC support vector machine from the LibSVM library is trained. We do 10-fold cross-validation per kernel, which is repeated 10 times with different randomization seeds. Within each fold, the \( d \) and \( C \) parameter is optimized, again using 10-fold cross-validation. The depth parameter \( d \) is optimized from the range: 2, 4, 6.\(^8\) For \( C \) we use the range: 1, 10, 10\(^2\), 10\(^3\), which is dynamically extended by a factor 10 if the best performance is achieved on the edge (e.g. 1 or 10\(^3\)) of the range.

\(^8\)We use only even settings since one triple \((s, p, o)\) consists of two edges in our graph representation.
Results. Table 2 shows the results for this experiment (and the following two) in terms of average accuracy, with the standard deviation shown in parentheses. The table shows the neighborhood extraction method (Graph, Tree or Direct) horizontally versus the different kind of substructures (BoL, Walks, Subtrees) vertically. The kernels that have the Root constraint are shown in a separate column, together with the Intersection SubTree (IST) and Intersection Partial SubTree (IPST) kernels. The best scores overall, and those scores that have no significant difference with those scores under a paired t-test with \( p < 0.05 \), are indicated in a bold type face. The best score per column, and those that do not differ significantly, are shown in italic type face. The computation of the GraphWalks kernel under the \( d = 6 \) setting did not finish in time, so for the GraphWalks kernel we optimized \( d \) from the range: 2,4.

<table>
<thead>
<tr>
<th>Affiliation</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
<th>Mutagenic</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoL</td>
<td>0.875(0.014)</td>
<td>0.909(0.003)</td>
<td>BoL</td>
<td>0.948(0.006)</td>
<td>0.951(0.004)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtrees</td>
<td>0.888(0.007)</td>
<td>0.912(0.004)</td>
<td>0.886(0.009)</td>
<td>Subtrees</td>
<td>0.914(0.005)</td>
<td>0.936(0.009)</td>
<td>0.914(0.005)</td>
</tr>
<tr>
<td>Walks</td>
<td>0.879(0.008)</td>
<td><strong>0.920(0.007)</strong></td>
<td>0.895(0.012)</td>
<td>Walks</td>
<td>0.914(0.004)</td>
<td>0.923(0.008)</td>
<td>0.914(0.004)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lithogenesis</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
<th>Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoL</td>
<td>0.859(0.011)</td>
<td><strong>0.822(0.013)</strong></td>
<td>Subtrees</td>
<td>0.553(0.011)</td>
</tr>
<tr>
<td>Subtrees</td>
<td><strong>0.861(0.014)</strong></td>
<td>0.844(0.013)</td>
<td>0.864(0.013)</td>
<td>Walks</td>
</tr>
<tr>
<td>Walks</td>
<td>0.826(0.016)</td>
<td>0.812(0.021)</td>
<td>0.795(0.018)</td>
<td>IPST</td>
</tr>
<tr>
<td></td>
<td>IST</td>
<td>0.827(0.012)</td>
<td>0.929(0.001)</td>
<td>0.804(0.020)</td>
</tr>
</tbody>
</table>

Slightly better performance was achieved in the no inference settings, so those results are shown in the table. The TreeWalks kernel achieves the overall best performance. The bag of labels (BoL) kernels, especially TreeBoL also show very good performance. The kernels that only allow substructures with the Root, especially the Subtrees variant, and the IntersectionSubTree kernels clearly show the worst performance.

4.1.2. Mutagenic Prediction

For our next experiment we use the MUTAG dataset, which is distributed as an example dataset for the DL-Learner\(^9\) [37] toolkit. It contains information about 340 complex molecules that are potentially carcinogenic, which is given by the ‘isMutagenic’ property, which is the property that we try to predict. The largest class in this dataset contains 211 of the 340 instances. The setup is the same as in the previous experiment. We remove the ‘isMutagenic’ property from the dataset.

Results. Table 2 shows the results for this experiment from the no inference setting. As in the affiliation prediction experiment, for the GraphWalks kernel

\(^9\)http://dl-learner.org
we optimized from the range: 2, 4. From the results it is very clear that the BoL kernels show the best performance.

4.1.3. Lithogenesis Prediction

For our next experiment we use a dataset from the British Geological Survey (BGS)\textsuperscript{10}, as we did earlier in [9]. This dataset contains information about geological measurements in the form of ‘Named Rock Units’. These named rock units have a number of properties to predict, one of them is the lithogenesis property, for which we take the four largest classes, leading to 163 instances.\textsuperscript{11} The setup is the same as in the previous two experiments. We remove the lithogenesis property and its inverse property from the dataset during training and testing.

Results. The results are given in Table 2. There are a number of good scores. The Subtrees kernels show the best performance. However, the GraphBoL kernel achieves a similar score. What is interesting is that the Walks kernels clearly perform worse than the BoL and Subtrees kernels. Like in the affiliation prediction experiment, the Root and the IntersectionSubTree kernels are the worst performers.

4.1.4. Theme Prediction

For the fourth experiment we again use the BGS dataset. This time we try to predict the geological theme. The dataset contains around 12000 named rock units with a geological theme. We use the three largest classes.\textsuperscript{12} The data is highly skewed, since the largest class contains around 10000 instances.

The experimental setup is similar to the two experiments above. However, we take 10 random subsets of 5% of the data and use 5-fold cross-validation. Furthermore, we use LibLINEAR to train support vector machines directly on the feature vector representations. We therefore do not include the IntersectionSubTree kernels, which do not have such a representation. Their exclusion is further justified by their weak (and similar to the Root kernels) performance in the previous two tasks.

Results. Table 3 contains the results for the setting with triple-store inferencing. Again, for the GraphWalks we used the range: 2, 4. The TreeSubtrees has the best performance. However, the TreeBoL kernel achieves very similar performance. Again, the Root kernels clearly perform worst.

\textsuperscript{10}http://data.bgs.ac.uk/
\textsuperscript{11}We expanded this experiment to four classes from two in the original version in [9] to make it more discriminative.
\textsuperscript{12}We expanded this experiment to three classes from two in the original version in [9] to make it more discriminative.
Table 3: Results for the Theme and Category Prediction experiments.

<table>
<thead>
<tr>
<th></th>
<th>Theme</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Graph</td>
<td>Tree</td>
<td>Direct</td>
<td>Root</td>
</tr>
<tr>
<td>BoL</td>
<td>.985</td>
<td>.992</td>
<td>.990</td>
<td>.625</td>
</tr>
<tr>
<td>Subtrees</td>
<td>.988</td>
<td>.993</td>
<td>.990</td>
<td>.984</td>
</tr>
<tr>
<td>Walks</td>
<td>.988</td>
<td>.990</td>
<td>.986</td>
<td>.984</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Category</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Graph</td>
<td>Tree</td>
<td>Direct</td>
<td>Root</td>
</tr>
<tr>
<td>BoL</td>
<td>.920</td>
<td>.930</td>
<td>.931</td>
<td>.519</td>
</tr>
<tr>
<td>Subtrees</td>
<td>.931</td>
<td>.931</td>
<td>.931</td>
<td>.931</td>
</tr>
<tr>
<td>Walks</td>
<td>.919</td>
<td>.919</td>
<td>.916</td>
<td>.917</td>
</tr>
</tbody>
</table>

4.1.5. Category Prediction

For our fifth classification experiment we use a dataset from the Amsterdam Museum [38], which contains information about around 70000 artifacts in the museum’s collection. Each artifact belongs to one of 18 categories, we try to predict this category for the largest 17 classes. Next to the category relation we also remove the highly correlated materials relation from the data during training and testing.

Similar to the previous experiment we perform classification on 10 random subsets, with 5% of the full dataset. We only use the range: 2, 4 for the depth parameter, since most kernels were difficult to compute for $d = 6$ because of very large instance graphs/trees. Again, we use LibLINEAR.

Results. We give the results in Table 3 for the setting with inferencing. Very clearly the best performance is achieved by the Subtrees kernels in all settings. Once again, the Root kernels show the worst performance.

4.1.6. Discussion

In three of the five experiments the best performing kernel is one of the Subtrees variants, only in the affiliation prediction experiment does the TreeWalks kernel perform better. However, the improvement of using the more complex kernels over the bag of labels (BoL) kernels is not large, for two experiments the BoL kernels achieve a similar score and in one experiment they are even clearly the best performers.

In all experiments the kernels that consider only substructures with the root, i.e. the Root kernels and both IntersectionSubTree kernels, are the worst performers. For a large number of settings they are outperformed by the simpler BoL kernels.

The five experiments show no clear preference for one of the three extraction methods (Graph, Tree and Direct). However, the Tree kernels clearly perform the best in the affiliation prediction experiment, which is interesting, because the difference in size between the graph and tree representation for that dataset is the largest. For the ‘no inference’ setting for which the results are shown, for depth 6 the trees have 5 times more edges than the graphs.
4.2. Classification with Removing Low Frequency Labels

For our experiments with the minFreq kernels we use the same classification tasks as above. In addition to optimizing over the $d$ parameter we also optimize over the minimal frequency parameter $\theta$, for which we used different ranges for different experiments. For the affiliation and theme prediction experiments we used: 0, 1, 2, 4, 8, 16, for lithogenesis prediction: 0, 2, 4, 8, 16, 32, for category prediction: 0, 1, 2, 4, 8, and finally for mutagenic: 0, 16, 32, 64, 128, 256, the mutagenic task really benefits from the higher thresholds. For the Subtrees kernels we tested with using both ‘sets’ and ‘multisets’ during the Weisfeiler-Lehman relabeling. In turned out that only in the lithogenesis prediction task switching to sets was beneficial. Also, in these experiments we do not set the label of the instances vertices to the same label, since the MinFreq modification generalizes this idea. The rest of the experimental setup is unchanged.

4.2.1. Results

The results for the 5 experiments are given in Table 4. With $+$ or $-$ a positive, resp. negative, significant difference, using a paired t-test with $p < 0.05$, with the corresponding score in the original experiment (i.e. the corresponding non MinFreq kernel) is given. The Subtrees kernels benefit more from removing the low frequency labels than the Walks kernels. Especially noteworthy is the improvement in the mutagenic prediction task, where all kernels improve. They showed clearly worse performance than the BoL kernels in the original experiment, but the performance is closer (and in two cases better) now.

<table>
<thead>
<tr>
<th>Affiliation</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
<th>Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtrees</td>
<td>.895$^+$(.009)</td>
<td>.914$(.006)$</td>
<td>.900$^+$(.010)</td>
<td>.581$^+$(.016)</td>
</tr>
<tr>
<td>Walks</td>
<td>.839$^-$(.013)</td>
<td>.918$(.006)$</td>
<td>.902(.007)</td>
<td>.828$^-$(.018)</td>
</tr>
<tr>
<td>Mutagenic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtrees</td>
<td>.937$^+$(.014)</td>
<td>.952$^+$(.007)</td>
<td>.958$^+$(.006)</td>
<td>.668$^+$(.005)</td>
</tr>
<tr>
<td>Walks</td>
<td>.936$^-$(.007)</td>
<td>.943$^+$(.007)</td>
<td>.944$^+$(.005)</td>
<td>.936$^+$(.006)</td>
</tr>
<tr>
<td>Lithogenesis</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtrees</td>
<td>.882$^+$(.011)</td>
<td>.863$^+$(.012)</td>
<td>.881$^+$(.009)</td>
<td>.593$^+$(.017)</td>
</tr>
<tr>
<td>Walks</td>
<td>.809$^-$(.020)</td>
<td>.795(.024)</td>
<td>.788(.020)</td>
<td>.764$^-$(.021)</td>
</tr>
<tr>
<td>Theme</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtrees</td>
<td>.990(.004)</td>
<td>.995$^+$(.002)</td>
<td>.990(.003)</td>
<td>.676$^+$(.042)</td>
</tr>
<tr>
<td>Walks</td>
<td>.984(.006)</td>
<td>.986$^-$(.003)</td>
<td>.984(.004)</td>
<td>.980(.008)</td>
</tr>
<tr>
<td>Category</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtrees</td>
<td>.935$^+$(.003)</td>
<td>.933$^+$(.003)</td>
<td>.933$^+$(.003)</td>
<td>.498$^-$(.024)</td>
</tr>
<tr>
<td>Walks</td>
<td>.923$^+$(.005)</td>
<td>.921(.004)</td>
<td>.924$^+$(.003)</td>
<td>.913$^-$(.006)</td>
</tr>
</tbody>
</table>

Table 4: Results for the 5 experiments using the MinFreq kernels.
4.2.2. Discussion

In all five experiments the MinFreq modification of the algorithms has a positive effect. Especially the variants of the Subtrees kernels achieve better performance, albeit relatively marginal in case of the theme prediction task. The preferred extraction method does not really change for the MinFreq kernels.

4.3. Classification with Hub Removal

For our hub removal experiments we again perform the same classification tasks. The original kernels (i.e. without the MinFreq modifications) are now computed on graphs with hubs removed, using Algorithm 3.5, for the following settings of the minimum frequency $k$: 10, 20, 40, 80, 160, $\infty$. Note that the $\infty$ setting means that the graph is unchanged. During training of the support vector machine we now optimize $k$ as well as $d$ and $C$.

4.3.1. Results

Table 5 gives the results for the affiliation, mutagenic and lithogenesis prediction experiments. The $+$ and $-$ have the same meaning as in the results table for the MinFreq experiments. For the affiliation prediction experiment we see a clear improvement for all the Tree kernels, especially the TreeSubtrees kernel performs clearly better. For the mutagenic prediction experiment all kernels improve, especially the Walks kernels are interesting, since they are now the best performers. Finally, in the lithogenesis experiment we see a clear negative impact of hub removal, with the GraphWalks and GraphSubtrees kernels drastically decreasing in performance.

Table 5: Results for the Affiliation, Mutagenic and Lithogenesis experiments using Hub Removal.

<table>
<thead>
<tr>
<th>Affiliation</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
<th>Mutagenic</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoL</td>
<td>.886 ($\pm .017$)</td>
<td>.931 ($\pm .008$)</td>
<td>BoL</td>
<td>.967 ($\pm .006$)</td>
<td>.961 ($\pm .005$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtrees</td>
<td>.896 ($\pm .014$)</td>
<td>.936 ($\pm .008$)</td>
<td>.895 ($\pm .014$)</td>
<td>Subtrees</td>
<td>.962 ($\pm .004$)</td>
<td>.962 ($\pm .006$)</td>
<td>.962 ($\pm .004$)</td>
</tr>
<tr>
<td>Walks</td>
<td>.884 ($\pm .016$)</td>
<td>.932 ($\pm .008$)</td>
<td>.920 ($\pm .010$)</td>
<td>Walks</td>
<td>.966 ($\pm .007$)</td>
<td>.963 ($\pm .009$)</td>
<td>.965 ($\pm .007$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lithogenesis</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
<th>Root</th>
<th>Aff.</th>
<th>Mutag.</th>
<th>Litho.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoL</td>
<td>.859 ($\pm .012$)</td>
<td>.839 ($\pm .016$)</td>
<td>Subtrees</td>
<td>.663 ($\pm .011$)</td>
<td>.944 ($\pm .009$)</td>
<td>.790 ($\pm .010$)</td>
<td></td>
</tr>
<tr>
<td>Subtrees</td>
<td>.640 ($\pm .015$)</td>
<td>.839 ($\pm .017$)</td>
<td>.841 ($\pm .019$)</td>
<td>Walks</td>
<td>.892 ($\pm .020$)</td>
<td>.966 ($\pm .006$)</td>
<td>.837 ($\pm .020$)</td>
</tr>
<tr>
<td>Walks</td>
<td>.671 ($\pm .018$)</td>
<td>.844 ($\pm .010$)</td>
<td>.838 ($\pm .023$)</td>
<td>IPST</td>
<td>.867 ($\pm .015$)</td>
<td>.941 ($\pm .003$)</td>
<td>.795 ($\pm .015$)</td>
</tr>
<tr>
<td>IST</td>
<td>.866 ($\pm .014$)</td>
<td>.947 ($\pm .004$)</td>
<td>.769 ($\pm .019$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The results for the theme and category prediction experiments are given in Table 6. Hub removal has almost no influence on performance, apart from quite substantially increasing the performance of the TreeSubtreesRoot kernel.

4.3.2. Discussion

Only in the first three experiments do we see a clear influence of the hub removal. Especially in the affiliation and mutagenic prediction task it leads to improved performance for a large number of kernels. For the lithogenesis prediction experiment the influence is more mixed, performance is increased for some
kernels, but also severely decreased for two kernels. For the two experiments on larger dataset hub removal has very little influence.

Our hub removal strategy can, in principle, be combined with the MinFreq kernels and this could potentially improve performance further. However, it also leads to the combined optimization of the parameters $\theta$ and $k$, substantially increasing training time.

4.4. Graphs without Labels

Even though the kernels that count more complex graph features (Walks and especially Subtrees) often show better performance than the baseline BoL kernels, the difference is smaller than might be expected. To get a feeling for the importance of graph structure in RDF graphs with respect to the classification tasks presented, we performed the first three of the above experiments (affiliation, mutagenic and lithogenesis prediction) on graphs with the vertex labels removed (i.e. all of them are set to the same label).

4.4.1. Results

The results for these experiments are given in Table 7. Note that the BoL kernels in all three experiments now perform on chance level.\(^{13}\) In the affiliation and lithogenesis experiments, the Subtrees kernels are clearly the best performers, whereas there is less difference between the Walks and Subtrees in the mutagenic experiment.

4.4.2. Discussion

These three experiments clearly show that there is information in the structure of the RDF graphs, as well as the labels. Especially the Subtrees kernels are able to exploit this information. However, the amount of information that the graph structure contains on top of the label information is apparently not very large. Since in the regular experiments, the difference between the BoL kernels and the Subtrees kernels is very small.

\(^{13}\)Their accuracy is the same as fraction of the largest class in the dataset.
### Table 7: Results for the Affiliation, Mutagenic and Lithogenesis experiments with labels removed.

<table>
<thead>
<tr>
<th>Affiliation</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
<th>Mutagenic</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoL</td>
<td>.412(0.000)</td>
<td>.412(0.000)</td>
<td></td>
<td>BoL</td>
<td>.621(0.000)</td>
<td>.621(0.000)</td>
<td></td>
</tr>
<tr>
<td>Subtrees</td>
<td>.844(0.009)</td>
<td>.884(0.018)</td>
<td>.862(0.009)</td>
<td>Subtrees</td>
<td>.708(0.016)</td>
<td>.726(0.012)</td>
<td>.715(0.016)</td>
</tr>
<tr>
<td>Walks</td>
<td>.549(0.014)</td>
<td>.698(0.013)</td>
<td>.647(0.013)</td>
<td>Walks</td>
<td>.689(0.009)</td>
<td>.621(0.000)</td>
<td>.703(0.009)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lithogenesis</th>
<th>Graph</th>
<th>Tree</th>
<th>Direct</th>
<th>Aff.</th>
<th>Mutag.</th>
<th>Litho.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoL</td>
<td>.571(0.000)</td>
<td>.571(0.000)</td>
<td></td>
<td>Subtrees</td>
<td>.514(0.011)</td>
<td>.624(0.016)</td>
</tr>
<tr>
<td>Subtrees</td>
<td>.809(0.019)</td>
<td>.689(0.024)</td>
<td>.740(0.016)</td>
<td>Walks</td>
<td>.692(0.011)</td>
<td>.723(0.007)</td>
</tr>
<tr>
<td>Walks</td>
<td>.657(0.014)</td>
<td>.569(0.004)</td>
<td>.581(0.017)</td>
<td>IST</td>
<td>.403(0.018)</td>
<td>.618(0.008)</td>
</tr>
</tbody>
</table>

### 4.5. Computation Time

Finally, we present the computation time of a representative number of kernels for three different datasets. For each of those datasets we compute the kernel on a number of fractions of the entire dataset extracted for $d = 6$, with inferencing. Note that the implementations for the two bag of labels (BoL) kernels are not as efficient as possible, since they are essentially $h = 0$ Subtrees kernels. They are still included as a reference though. Note that the computation time shown is for the substructure counting (e.g. the Weisfeiler-Lehman algorithm) and kernel matrix computation steps of the kernels, all of the kernels also have an instance extraction step (either to graphs or trees, or to maps that track the vertices and edges for an instance in case of the Direct kernels). This step is similar in terms of computation time for all the kernels. So, the substructure counting step is what determines the differences in computation time between the kernels.

#### 4.5.1. Lithogenesis Prediction

Figure 3 shows the computation times for the lithogenesis prediction dataset. This is the only figure which includes the GraphWalks kernel. Already on this small dataset it takes far more time to compute than the other kernels. The rest of the kernels are close together, only the two BoL kernels are clearly faster.

#### 4.5.2. Affiliation Prediction

The computation time for the affiliation prediction dataset is given in Figure 4. We can see that for larger fractions the DirectSubtrees kernel more and more outperform the GraphSubtrees kernel as predicted, as does the TreeSubtrees kernel. The Walks are substantially slower on this dataset, since there are a large number of cycles in the RDF dataset. Note that for this dataset, GraphWalks only finished computation for the first few fractions, with times that would be far off this figure with a factor 10 or more.
Figure 3: Runtimes (msecs) for different kernels on the Lithogenesis Prediction dataset, with errors bars indicating the standard deviation.

Figure 4: Runtimes (msecs) for different kernels on the Affiliation Prediction dataset, with errors bars indicating the standard deviation.
4.5.3. Theme Prediction

For the theme prediction dataset computation times are given in Figure 5. Since we use feature vectors in this experiment, the times are for computing the feature vectors. Again, as the fractions become larger, the difference between the DirectSubtrees and GraphSubtrees kernel grows. This dataset contains far less cycles and therefore the TreeSubtrees is very close in computation time to DirectSubtrees. Again, GraphWalks only finished computation for the first few fractions, but the times would be far off the figure.

![Figure 5: Runtimes (msecs) for different kernels on the Theme Prediction dataset, with errors bars indicating the standard deviation.](image)

4.5.4. Discussion

From the results of the classification experiment and the computation times, we can see that the DirectWalks kernel makes the GraphWalks kernel computationally viable. The large difference in computation time is due to the exponential growth in the number of walks in graphs with cycles. Hence the fact that the GraphWalks kernel is already hard to compute on the small dataset used in the affiliation prediction experiment.

The DirectSubtrees algorithm also speeds up the GraphSubtrees kernel by around a factor 3 for the full affiliation and theme prediction datasets. And as we can see from the figures this factor grows when datasets become larger.

5. Conclusions and Future Work

We have presented a framework for substructure counting graph kernels for RDF data. This framework systematically covers most of the graph kernels introduced for RDF and also provides a number of new kernels. The definitions
include kernel variants that are computed directly on the RDF graph. We detailed the adaptation of the Weisfeiler-Lehman graph kernel [8], needed to compute the subtree counting kernels, to ensure that identical subtrees are not repeatedly counted. Furthermore, we introduced two strategies to deal with the unique label problem of RDF graphs. The first strategy ignores vertex labels that have a low frequency of occurrence among the instances and the second strategy involves removing hubs to simplify RDF graphs and creating more general vertex labels.

We tested these methods in a number of classification experiments. These experiments show that the graph kernels that count subtrees overall give the best performance and are the best at exploiting the graph structure in RDF data. However, simple bag of labels baseline kernels also perform very well, while being substantially faster to compute. The kernels that count only substructures that include the root vertex achieve the worst performance. They are often outperformed by the simple bag of labels kernels.

The direct kernel definitions that we introduce provide a substantial computational speed up. In case of the kernel that counts walks this speed up is so large that it makes the kernel viable to compute and use. Furthermore the direct definitions do not hamper performance.

Our low frequency label removal strategy has a positive impact on performance in all of our five experiments. Especially the kernels that count subtrees are positively impacted.

Our hub removal strategy generally improves performance on two of the five datasets. On the two largest datasets hub removal has little impact (nor positive, nor negative). A future research direction would be to see if this holds for larger RDF datasets in general. Furthermore, an interesting direction for future work is to combine the hub removal and low frequency label removal strategies.

Different RDF datasets can have a very different graph structure. The experimental results on our dataset with the most graph like structure (i.e. the most cycles), seems to suggest that kernels that use the instance tree representation work best on those datasets. In future work we want to get a better understanding of the influence of these structural difference in RDF graphs on the computation of the kernels and their performance. This could help in determining which kernel to use, without computing them all.

Link prediction in RDF graphs is an important problem, for which several approaches exists. Using ranking variants of Support Vector Machines [3, 4] the kernels defined in this paper could be used to perform link prediction. It would be interesting future research to see how such an approach compares to existing methods.

Currently, our kernels can only be used on RDF graphs that fit in main memory. However, the kernels discussed in this framework, and especially the kernels computed directly on the RDF graph, are suitable to implement on large graph processing platforms, such as GraphLab [39] and Pregel [40].
Acknowledgments

This publication was supported by the Dutch national program COMMIT and by the Netherlands eScience center. Great thanks go out to Peter Bloem for valuable discussions.

References


URL http://dl.acm.org/citation.cfm?id=1953048.2078187


URL http://doi.acm.org/10.1145/2623330.2623623

URL http://dx.doi.org/10.1109/ICDM.2014.83


35