A Fast and Simple Graph Kernel for RDF

Gerben Klaas Dirk de Vries[1] and Steven de Rooij[1,2]

[1] System and Network Engineering Group, Informatics Institute, University of Amsterdam, The Netherlands
g.k.d.devries@uva.nl
[2] Knowledge Representation and Reasoning Group, Department of Computer Science, VU University Amsterdam, The Netherlands
steven.de.rooij@gmail.com

Abstract. In this paper we study a graph kernel for RDF based on constructing a tree for each instance and counting the number of paths in that tree. In our experiments this kernel shows comparable classification performance to the previously introduced intersection subtree kernel, but is significantly faster in terms of computation time. Prediction performance is worse than the state-of-the-art Weisfeiler Lehman RDF kernel, but our kernel is a factor 10 faster to compute. Thus, we consider this kernel a very suitable baseline for learning from RDF data. Furthermore, we extend this kernel to handle RDF literals as bag-of-words feature vectors, which increases performance in two of the four experiments.

Keywords: Resource Description Framework (RDF), Graph Kernels, Intersection Tree Kernels

1 Introduction

Machine learning and data-mining for linked data is an emerging research field. Linked data is represented in the Resource Description Framework (RDF). Kernel methods [1, 2] are a popular method for handling structured data. For structured data in the form of graphs, graph kernels have been introduced, see for example [3] and [4].

Since RDF essentially represent a graph, graph kernel methods provide an attractive solution for machine learning on linked data. Currently there exists some research on graph kernels for RDF. In [5] two types of graph kernels for RDF are introduced and tested on a number of tasks. A faster variant for RDF of the general Weisfeiler-Lehman kernel for graphs is described in [6]. This kernel is shown to be both faster and better performing than the kernels in [5].

The advantage of using graph kernel techniques for RDF is that the technique is very generic and can be widely applied, as long as the data is stored as RDF. No understanding of the semantics of the data is required, although such knowledge can be used if available. For example, if the data is modeled using a

more expressive formalism such as OWL and RDFS than extra derived triples can be added to the RDF and this information can be exploited as well.

In [7] kernels are manually designed by selecting task relevant properties and concepts for the instances, and then incorporating these in the kernel measure. Other attempts at learning from semantic web data are based on description logic [8, 9]. These approaches are less generic and/or require more domain knowledge than the graph kernel method.

In this paper we introduce a simple graph kernel for RDF data based on constructing a tree for each instance, starting from the root, i.e. the instance, vertex, up to a defined depth. Then we count the number of paths in the tree starting from the root vertex. This kernel shows comparable classification performance to the intersection subtree kernel defined in [5]. However, our kernel is significantly faster in terms of computation time. Furthermore, the kernel is easily extended to treat RDF literals using bag-of-words vectors. Prediction performance is worse than the state-of-the-art Weisfeiler-Lehman RDF kernel in [6], but it is significantly faster to compute. Therefore, we consider this kernel as a suitable baseline for learning from RDF data.

The rest of this paper is structured as follows. In Sect. 2 we introduce our graph kernel and its extension to deal with literals. Section 3 contains our experimental evaluation of the kernel performance in terms of prediction and computation time. We end with some conclusions and suggestions for future work.

2 Intersection Tree Path Kernel

The graph kernel approach to learning from RDF is based on the idea that instances are represented by their subgraphs. In the classification task introduced in [7], and repeated in [5], the authors tried to predict the affiliation (i.e. research group) for people in a research institute. In this task the instances are people, and they are represented by the subgraphs (up to a certain depth) around their root nodes, i.e. the URIs that identify them. Predicting which research group a person belongs to can be performed by computing a kernel between the subgraphs and training a Support Vector Machine (SVM) with this kernel.

2.1 Algorithm

Below we present a kernel method for RDF graphs very similar to the intersection subtree kernels in [5]. As in that kernel, we construct the tree starting from each instance vertex, up to a certain depth d. We do this by extracting all paths from length d starting from the instance vertex. Even though we define them on graphs, these kernels are essentially tree kernels. See [10] and [1] for the most common tree kernels.

There are two main differences between the kernel defined here and the intersection subtree kernel in [5]. First, we count the paths starting from the root vertex, instead of the (partial) subtrees. Second, we explicitly compute feature vectors, instead of pairwise similarities. Computing the feature vectors

significantly improves computation time and allows the use of large-scale linear learning methods, such as LibLINEAR [11], when dealing with large amounts of instances. By taking the dot product of two feature vectors we essentially count the paths that two instance trees have in common, i.e. the paths in the intersection tree. Therefore we dub this kernel: Intersection Tree Path kernel.

Our method is presented in Algorithm 1. For each instance we compute a feature vector using the recursive function *processVertex*. This recursive function counts all the paths starting from the instance vertex up to depth d and creates a feature vector with these counts. Using the map *pathMap* we track the feature vector index of each unique occurring path. Furthermore, for efficiency it is useful to implement the feature vectors as sparse vectors, since the majority of the indices have zero value. Note that, as in the kernel in [5], we give the root vertices the same special label *rootID*.

We count paths instead of subtrees for two reasons. First, it leads to a very straightforward computation of the feature vectors, since we do not have to compute the full tree first to compute the subtrees. Furthermore, it allows for handling (string) literals separately in a simple manner, as we will see in Sect. 2.2.

Algorithm 1 Intersection Tree Path Kernel

Input a set of RDF triples R, a set of instances I and a max depth d_{max}
Output a set of feature vectors F corresponding to the instances I
Comments *pathMap* is a global map between paths of vertices and edges and integers

- set $pathIdx = 1$
- for each $i \in I$
  - create a new feature vector fv
  - do $processVertex(i, i, [], fv, d_{max})$
  - add fv to F

function $processVertex(v, root, path, fv, d)$
- if $d = 0$, return
- for each $(v, p, o) \in R$
  - if o is $root$, set $path = [path, p, rootID]$
  - else, $path = [path, p, o]$
  - if $pathMap(path)$ is undefined
    - set $pathMap(path) = pathIdx$
    - set $pathIdx = pathIdx + 1$
  - set $fv(pathMap(path)) = fv(pathMap(path)) + 1$
  - do $processVertex(o, root, path, fv, d - 1)$

Compared to the Weisfeiler-Lehman (WL) graph kernel used in [6] and the intersection graph kernel in [5] our tree kernel is more memory efficient, since it does not require a construction of the subgraph for each instance. As with the WL graph kernel, we also explicitly compute feature vectors.

2.2 Literals as Bag-of-Words

Literals in RDF can only occur as objects in triples and therefore have no outgoing edges in the RDF graph. They can contain different types of data, but they are often strings. In Algorithm 1 literals are treated as any other vertex and they are only considered equal if they contain exactly the same information. However, for (large) strings this approach can waste potentially useful information. Therefore we introduce an extension to the previous algorithm that treats literals as separate bag-of-words vectors.

Algorithm 2 is essentially Algorithm 1, with a different treatment of literal vertices. For each instance we now create a list of feature vectors *fvs*, which are filled by the function *processVertex*. If *processVertex* encounters a literal, then we add a special literal label to the path (similar to how root vertices are treated) and the function *processLiteral* is executed with that literal. First, *processLiteral* creates a term vector for the literal, counting all the terms/words occurring in that literal. To compare only literals that have the same path from the root vertex, we use the map *litMap* to track an index into the list of feature vectors *fvs* and put the term vector into *fvs* at that index. For simplicity we treat all literals as strings and we assume that one underlying dictionary is used when creating the term vectors, ensuring that each term has the same index in each vector. Furthermore, for ease of definition, we assume that the list of feature vectors *fvs* is long enough. Moreover, the individual term vectors have a fixed length that is large enough to fit all the possible terms, such that concatenating them results in vectors of the same length for every instance.

When working with text and term vectors it is standard to normalize term vectors by converting them to Term Frequency-Inverse Document Frequency (TF-IDF) vectors. We also apply this normalization to our computed feature vectors.

3 Experiments

In this section we present a number of experiments with the Intersection Tree Path (ITP) kernel that we presented above. The first goal of these experiments is to compare the performance of the ITP kernel with the Intersection SubTree kernel in [5]. The experiments in [6] show that there is very little difference in performance between the two Intersection SubTree kernels given in [5]. Therefore we only compare to the Intersection (Full) SubTree (IST) kernel, which performed slightly better. Secondly, we also compare the performance to the Weisfeiler-Lehman RDF (WL RDF) kernel from [6], which was the best performing kernel in that paper. We repeat the three prediction tasks presented in [6] and add one extra task. Since one of the reasons for introducing the Intersection Tree Path kernel is computation time, we also compare the runtimes of the kernels.

Algorithm 2 defines the Intersection Tree Path with Bag-of-Words kernel (ITP-BoW), a method for dealing with literals by treating them as a term vector. For comparison, we introduce a similar extension to the Weisfeiler-Lehman RDF

Algorithm 2 Intersection Tree Path With Bag-of-Words Kernel

Input a set of RDF triples R, a set of instances I and a max depth d_{max}
Output a set of feature vectors F corresponding to the instances I
Comments *pathMap* is a global map between paths of vertices and edges and integers

- set $pathIdx = 1$, $litIdx = 2$
- for each $i \in I$
  - create a list of new feature vectors fvs
  - do $processVertex(i, i, [], fvs, d_{max})$
  - concatenate all feature vectors in fvs to one feature vector fv
  - add fv to F

function $processVertex(v, root, path, fvs, d)$
- if $d = 0$, return
- for each $(v, p, o) \in R$
  - if o is $root$, set $path = [path, p, rootID]$
  - if o is a Literal, set $path = [path, p, literalID]$
  - else, set $path = [path, p, o]$
  - if $pathMap(path)$ is undefined
    - set $pathMap(path) = pathIdx$
    - set $pathIdx = pathIdx + 1$
  - set $fvs(1)(pathMap(path)) = fvs(1)(pathMap(path)) + 1$
  - if o is a Literal do $processLiteral(o, path, fvs)$
  - else do $processVertex(o, root, path, fvs, d - 1)$

function $processLiteral(literal, path, fvs)$
- create term vector tf for $literal$
- if $litMap(path)$ is undefined
  - set $litMap(path) = litIdx$
  - set $litIdx = litIdx + 1$

  add tf to the feature vector $fvs(litMap(path))$

kernel, dubbed Weisfeiler-Lehman RDF with Bag-of-Words (WL RDF-BoW). The Weisfeiler-Lehman algorithm iteratively rewrites vertex and edge labels by creating a new label using the labels of their neighborhoods. The counts of these rewritten labels introduce a feature vector, similar to the counts of the paths in ITP. To extend WL RDF with term vectors for literals we also introduce a special literal label and create a term vector for the original content of the literal. Just as in Algorithm 2 we need a list of feature vectors at each iteration of WL RDF. Instead of indexing these feature vectors using a map from paths to integers, we use a map from labels to integers, where the labels are the (rewritten) labels of the literal vertices at each iteration.

The Intersection SubTree is implemented as suggested in [5] and the RDF Weisfeiler-Lehman kernels as defined in [6]. As in [6] we test three subgraph extraction depth settings $(1, 2, 3)$. Also that paper shows that allowing RDFS

inferencing by the triple store produces better results, therefore we apply this setting.

We apply TF-IDF weighting to the ITP-BoW kernel, because that increases performance. However, the WL RDF-BoW kernel performs slightly better without TF-IDF.

All of the kernels and experiments where implemented in Java and the code is available online.[3] The Java version of the LibSVM [12] support vector machine library was used for prediction with the kernels and the SESAME[4] triple-store was used to handle the RDF data and do the RDFS inferencing. The term vectors were created using Java's text library. The experiments were run on an AMD X6 1090T CPU with 16 GB RAM.

3.1 Affiliation Prediction

For our first experiment we repeat the affiliation prediction experiment from [6], which was introduced in [7] and repeated in [5]. In this experiment data is used from the semantic portal of the AIFB research institute. The goal is to predict one of four affiliations for the people in the institute. The fifth affiliation is ignored, since it only has 4 members, leaving a total of 174 instances. Since the affiliations are known, the affiliation relation (and its inverse the employs relation) are removed from the RDF for training.

As we do for the ITP kernel, we set the label of the root vertex for each instance to an identical special root label for the WL RDF kernel. Furthermore, for the WL RDF kernel we test the number of iterations setting h with: $0, 2, 4, 6$. The intersection subtree kernel has the discount factor parameter λ which is set to best setting as reported in [5].

We use the C-SVC support vector machine algorithm from LibSVM to train a classifier for each kernel. Per kernel we do a 10-fold cross-validation which is repeated 10 times. Within each fold the C parameter is optimized from the range: $\{10^{-3}, 10^{-2}, 0.1, 1, 10, 10^2, 10^3\}$ by doing 10-fold cross-validation. The different classes are weighted with the inverse of their frequency. All kernels are normalized.

Table 1 presents the average accuracy and the average F1[5] scores for each class. The best scores and those that have no significant difference with these scores are indicated in bold typeface. The best scores for the two intersection tree path kernels, and those that have no significant difference, are indicated with an italic type face. A Student t-test with $p < 0.05$ is used as a significance test.

The RDF Weisfeiler-Lehman kernel clearly shows the best performance in this task. The intersection tree path kernel performs on a par with the intersection subtree kernel. Adding the bag-of-words representation of the literals to the kernels does not increase performance.

[3] `https://github.com/Data2Semantics/d2s-tools`

[4] `http://www.openrdf.org/`

[5] This is the average of the F1 scores for each class.

Table 1. Results for the affiliation prediction experiments.

depth	acc.	F1	acc.	F1	acc.	F1	acc.	F1
	Weisfeiler-Lehman RDF							
	$h = 0$		$h = 2$		$h = 4$		$h = 6$	
1	0.78	0.59	0.79	0.59	0.79	0.59	0.79	0.59
2	0.57	0.36	0.83	0.66	*0.82*	*0.63*	0.80	0.60
3	0.74	0.57	**0.91**	**0.81**	0.90	**0.80**	0.90	0.79
	Weisfeiler-Lehman RDF with Bag-of-Words							
	$h = 0$		$h = 2$		$h = 4$		$h = 6$	
1	0.72	0.52	0.74	0.54	0.72	0.52	0.71	0.51
2	0.76	0.57	0.85	0.68	*0.82*	*0.63*	0.81	0.61
3	0.77	0.60	**0.91**	**0.80**	0.90	**0.80**	0.90	0.79
	Intersection Tree Path				Intersection Tree Path with Bag-of-Words			
1	*0.81*	0.61			0.71	0.48		
2	0.78	0.57			0.76	0.58		
3	*0.81*	0.61			*0.82*	*0.63*		
	Intersection Sub Tree, $\lambda = 1$							
1	*0.81*	*0.61*						
2	0.78	0.57						
3	*0.81*	0.61						

3.2 Lithogenesis Prediction

The next experiment is a repeat of the lithogenesis prediction task in [6]. In this task data from the British Geological Survey[6] is used, which contains information about geological measurements in Britain. The things measured by this survey are 'Named Rocked Units'. For these named rock units we try to predict the lithogenesis property, for which the two largest classes have 93 and 53 instances. Again, triples related to this property are removed from the dataset. The experimental setup is similar to the first affiliation prediction experiment.

Table 2 presents the results. As in the previous experiment, the WL RDF kernel shows the best performance. Also, similar to the previous experiment, the intersection tree path kernel shows similar performance to the intersection subtree kernel. Adding the term vectors for the literals does increase performance somewhat, however, it is not significant.

3.3 Multi-Contract Prediction

For our third experiment we use data provided for task 2 of the linked data data-mining challenge 2013.[7] We use only the dataset provided as training data,[8]

[6] `http://data.bgs.ac.uk/`

[7] `http://keg.vse.cz/dmold2013/data-description.html`

[8] The test set labels are not available at the time of writing.

Table 2. Results for the lithogenesis prediction experiments.

depth	acc.	F1	acc.	F1	acc.	F1	acc.	F1
	Weisfeiler-Lehman RDF							
	$h = 0$		$h = 2$		$h = 4$		$h = 6$	
1	0.77	0.60	0.78	0.61	0.78	0.60	0.78	0.60
2	0.82	0.66	0.87	0.76	*0.87*	0.75	*0.87*	0.75
3	0.88	0.76	0.88	0.77	**0.91**	**0.82**	**0.90**	**0.81**
	Weisfeiler-Lehman RDF with Bag-of-Words							
	$h = 0$		$h = 2$		$h = 4$		$h = 6$	
1	0.88	0.77	0.88	0.78	0.88	0.77	0.88	0.77
2	0.89	0.78	0.90	0.80	0.90	0.80	0.89	0.79
3	0.88	0.77	0.88	0.77	**0.91**	**0.82**	**0.92**	**0.83**
	Intersection Tree Path				Intersection Tree Path with Bag-of-Words			
1	0.77	0.59			*0.84*	*0.69*		
2	*0.84*	*0.70*			*0.85*	*0.71*		
3	*0.85*	*0.72*			*0.85*	*0.71*		
	Intersection Sub Tree, $\lambda = 1$							
1	0.77	0.60						
2	0.84	*0.70*						
3	*0.85*	*0.72*						

which contains 208 instances of government contracts. Of these instances 40 are labeled multicontracts. The task is to predict whether a contract has the property that it is a multicontract. The experimental setup is identical to the two tasks presented above. There is one difference, for this dataset the performance of the WL RDF-BoW kernel could be significantly improved by applying TF-IDF normalization.

Table 3 shows the results for this experiment. Because the scores were not normally distributed very well, we used a MannWhitney U test as our significance test. This multicontract prediction task is difficult. The baseline for this task is already an accuracy of 0.81 and an F1 score of 0.50 and only somewhat higher scores are obtained. The best results are achieved by the bag-of-words kernels, achieving especially higher F1 scores than the non-bag-of-words kernels. Again, the intersection tree path kernel performs similar to the intersection subtree kernel.

3.4 Theme Prediction

As our final prediction experiment we replicate the geological theme prediction experiment in [6]. The task again uses the named rock units in the British Geological Survey data. Each named rock unit has an associated geological theme. In this task we try to predict that theme, which has two major classes of size 10020 and size 1377.

Table 3. Results for the multicontract prediction experiments.

depth	acc.	F1	acc.	F1	acc.	F1	acc.	F1
	Weisfeiler-Lehman RDF							
	$h = 0$		$h = 2$		$h = 4$		$h = 6$	
1	0.81	0.43	0.81	0.44	0.81	0.43	0.81	0.43
2	0.75	0.43	0.82	0.51	0.81	0.51	0.82	0.50
3	0.56	0.32	0.81	0.50	0.82	0.51	0.82	0.51
	Weisfeiler-Lehman RDF with Bag-of-Words (TF-IDF)							
	$h = 0$		$h = 2$		$h = 4$		$h = 6$	
1	0.83	0.51	0.82	0.47	0.82	0.46	0.82	0.46
2	**0.85**	**0.58**	0.84	0.57	0.84	0.56	0.83	0.54
3	0.84	**0.57**	0.81	0.49	0.81	0.49	0.81	0.49
	Intersection Tree Path				Intersection Tree Path with Bag-of-Words			
1	0.81	0.44			0.82	0.49		
2	0.85	0.54			**0.85**	**0.57**		
3	0.84	0.54			0.81	0.49		
	Intersection Sub Tree, $\lambda = 1$							
1	0.81	0.44						
2	0.84	0.54						
3	0.84	0.52						

The experimental setup of this experiment is similar to the previous three. However, instead of repeating the task 10 times on the whole dataset, we take 10 random subsets of 10% of the data. Furthermore, we use the LibLINEAR [11] algorithm for the kernels that directly compute feature vectors. Also, we reduce computation time by using 5-fold cross-validation.

Table 4 present the results. We again use a MannWhitney U test as our significance test. The best performance is achieved by the intersection tree path kernel, however, the scores do not differ significantly from the best scores for the two WL RDF kernels. Again, the regular intersection tree path kernel performance is similar to the intersection sub tree kernel.

3.5 Runtimes

To test the runtimes of the different kernels we computed them with the highest settings (depth 3 and $h = 6$) on the dataset of the theme prediction experiment. The runtimes are measured on different fractions, 0.1 to 1, of the dataset.[9] We do not include the kernels with the bag-of-words representation of the literals, since we did not fully optimize the performance of the term vectors, and hence the comparison would not be fair. For the intersection path tree and the WL

[9] The fraction 1 is 10% of the total dataset, since we use the datasets from the previous experiment.

Table 4. Results for the theme prediction experiments.

depth	acc.	F1	acc.	F1	acc.	F1	acc.	F1
	Weisfeiler-Lehman RDF							
	$h = 0$		$h = 2$		$h = 4$		$h = 6$	
1	0.979	0.905	0.981	0.911	0.981	0.913	0.981	0.913
2	0.982	0.924	0.992	0.964	0.992	0.962	0.991	0.957
3	0.977	0.907	**0.995**	**0.979**	**0.995**	**0.979**	**0.995**	0.977
	Weisfeiler-Lehman RDF with Bag-of-Words							
	$h = 0$		$h = 2$		$h = 4$		$h = 6$	
1	0.985	0.933	0.990	0.956	0.991	0.958	0.991	0.958
2	0.991	0.960	0.995	0.976	0.994	0.974	0.994	0.970
3	0.990	0.956	**0.996**	**0.981**	0.995	0.977	**0.995**	**0.978**
	Intersection Tree Path				Intersection Tree Path with Bag-of-Words			
1	0.986	0.933			0.990	0.954		
2	0.993	0.968			**0.996**	**0.983**		
3	0.991	0.958			**0.996**	**0.981**		
	Intersection Sub Tree, $\lambda = 1$							
1	0.985	0.931						
2	0.994	0.970						
3	0.991	0.958						

RDF kernel we show the computation time for computing the feature vectors and for computing the kernel matrix, which includes the feature vectors.

Figure 1 presents the results. The intersection tree path feature vectors are the fastest to compute by a large margin, around 10 times faster than the WL RDF feature vectors. The computation time of the intersection subtree kernel scales quadratically with the number of instances and is also clearly slower than the WL RDF feature vectors for more instances. It is interesting to see that for both the WL RDF kernel and the ITP kernel the computation of the kernel (i.e. taking the dot product of the feature vectors) becomes the largest component of the computation time for large datasets. The computation of the feature vectors scales linearly with the amount of instances, whereas the dot product computation scales quadratically. For large numbers of instances with sparse feature vectors, linear learning algorithms (e.g. LibLINEAR), which work directly on feature vectors, are more efficient than algorithms working with a kernel/dot product (e.g. LibSVM). So for learning with larger numbers of instances there is a large computational benefit in having an explicit feature vector representation, which is both faster to compute and faster to learn with.

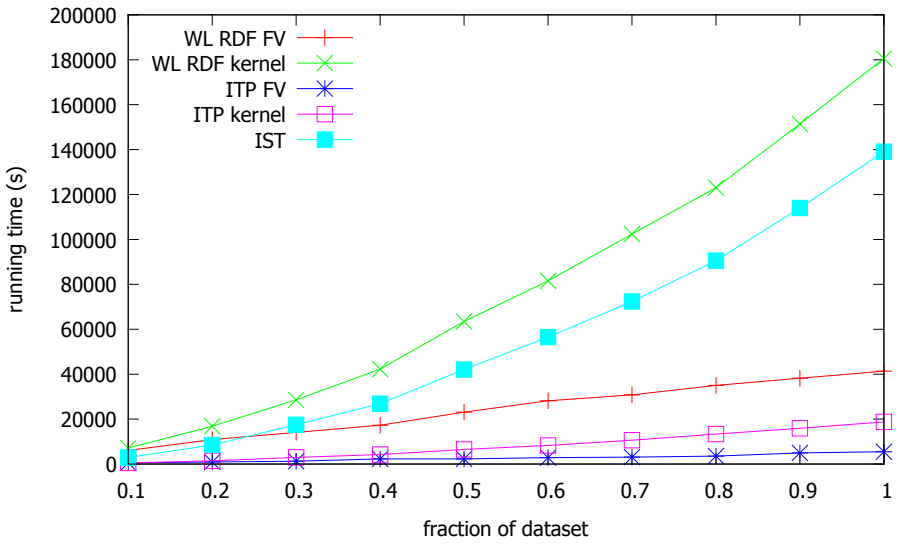

Fig. 1. Runtimes for the Weisfeiler-Lehman RDF feature vectors, the Weisfeiler-Lehman RDF kernel, the intersection tree path feature vectors, the intersection tree path kernel and the intersection subtree kernel.

4 Conclusions and Future Work

We presented a graph kernel for RDF based on counting the paths in the tree starting from the root instance vertex. In our experiments this kernel has similar performance to the previously introduced intersection subtree kernel, but is much faster to compute. However, in three experiments, the Weisfeiler-Lehman RDF kernel shows better performance. In one experiment the intersection tree path kernel achieves similar scores to the WL RDF kernel. Even though the WL RDF kernel also computes the feature vectors directly, its computation time is around a factor 10 longer than the intersection tree path kernel. Therefore, we think that our kernel is a good baseline kernel for learning from RDF.

Furthermore, we introduced an extension to the intersection tree path kernel to treat literals as separate bag-of-words vectors. In two experiments this does indeed increase performance over the regular intersection tree path kernel.

More research is required into what property of RDF datasets makes that the Weisfeiler-Lehman kernel outperforms the tree type kernels by a large margin in some tasks, but not in others. Manual inspection of the datasets used suggests that the multicontract prediction dataset is relatively hierarchical or tree-like in nature, whereas the affiliation prediction dataset has a more graph-like structure, with more cycles. This could be an explanation of the difference in performance.

We also want to investigate the selection of subsets of predicates and objects to consider during the construction of the trees. There are a large number of predicates and objects that occur very infrequently and are potentially better

treated as the same predicate/object or just removed from the tree. Given the fast computation time and relatively low memory requirement of the intersection tree path kernel an obvious next step for research it to test its feasibility on very large datasets, with millions of triples or more.

Acknowledgments This publication was supported by the Dutch national program COMMIT. We thank the authors of [5] for the AIFB dataset.

References

1. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, New York, NY, USA (2004)
2. Schölkopf, B., Smola, A.J.: Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge, MA, USA (2001)
3. Vishwanathan, S.V.N., Schraudolph, N.N., Kondor, R.I., Borgwardt, K.M.: Graph kernels. Journal of Machine Learning Research **11** (2010) 1201–1242
4. Shervashidze, N., Schweitzer, P., van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-lehman graph kernels. J. Mach. Learn. Res. **12** (November 2011) 2539–2561
5. Lösch, U., Bloehdorn, S., Rettinger, A.: Graph kernels for rdf data. In Simperl, E., Cimiano, P., Polleres, A., Corcho, Ó., Presutti, V., eds.: ESWC. Volume 7295 of Lecture Notes in Computer Science., Springer (2012) 134–148
6. de Vries, G.K.D.: A fast approximation of the weisfeiler-lehman graph kernel for rdf data. In Blockeel, H., Kersting, K., Nijssen, S., Zelezný, F., eds.: ECML/PKDD. (2013)
7. Bloehdorn, S., Sure, Y.: Kernel Methods for Mining Instance Data in Ontologies. The Semantic Web (2008) 58–71
8. Fanizzi, N., d'Amato, C.: A declarative kernel for ALC concept descriptions. In Esposito, F., Ras, Z.W., Malerba, D., Semeraro, G., eds.: Foundations of Intelligent Systems, 16th International Symposium,. Volume 4203 of Lecture Notes in Computer Science., Springer, Berlin–Heidelberg, Germany (2006) 322–331
9. Fanizzi, N., D'Amato, C., Esposito, F.: Statistical learning for inductive query answering on owl ontologies. In: Proceedings of the 7th International Conference on The Semantic Web. ISWC '08, Berlin, Heidelberg, Springer-Verlag (2008) 195–212
10. Collins, M., Duffy, N.: Convolution kernels for natural language. In: Advances in Neural Information Processing Systems 14, MIT Press (2001) 625–632
11. Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research **9** (2008) 1871–1874
12. Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology **2** (2011) 27:1–27:27 Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.