Efficient Graph Kernels for RDF data using Spark

Bernhard Japes[1] and Shinho Kang[2]

[1] Informatik III, Universität Bonn, Germany
bernhard.japes@uni-bonn.de
[2] Informatik III, Universität Bonn, Germany
shinho.kang@uni-bonn.de

Abstract

In this paper we study the application of graph kernels for RDF data using the popular Apache Spark[1] engine in combination with the SANSA-Stack[2] data flow utilities. We focus on an implementation of the Intersection Tree Path (ITP) Kernel, published by Gerben Klaas Dirk de Vries and Steven de Rooij in [2], that is based on the concept of constructing a tree for each instance and counting the number of paths in that tree.

1 Introduction

The increasing availability of structured data and the rise of the semantic web pose new challenges for machine learning and data mining. As an official standard, the *Resource Description Framework* (RDF) is commonly used to represent those graphs, which led to research on how to use the RDF structure to predict links and labels of instances efficiently. Most of the current approaches to mining structured graph-data focus on specific semantic properties and are individually designed for different problems [4, 3].

Kernels, however, have already been proven to be useful as a much more flexible approach for Pattern Analysis in different areas [5], which resulted in further research on specific graph kernels for RDF. The main drawback of most of these graph kernels, including the state-of-the-art *Weisfeiler-Lehman* (WL) RDF kernel, is their computation time as shown in [1]. In [2] Gerben Klaas Dirk de Vries and Steven de Rooij present a *Fast and Simple Graph Kernel for RDF* with just a slightly worse prediction performance than the WL graph kernel, but the huge upside of being 10 times faster in practice. Their idea of a fast and simple, but scalable kernel also seems to be promising for big data applications. However several adaptions of their algorithm are required to ensure consistent computations on distributed data sets using the Apache Spark engine.

2 Approach

The graph kernel presented in [2] is based on the idea that instances are represented by their subgraphs. This assumption implies that we should be able to explicitly compute a feature vector for each instance by constructing a tree, starting from the instance vertex up to a certain depth d, and counting the paths. Now taking the dot product of two feature vectors is essentially the intersection of both trees. The original pseudo-code for the so called Intersection Tree Path (ITP) Kernel is given in Algorithm 1.

[1]http://spark.apache.org
[2]http://www.sansa-stack.net

Algorithm 1: The Intersection Tree Path (ITP) Kernel as introduced in [2]

```
**Data**: a set of RDF triples $R$, a set of instances $I$ and a max depth $d_{max}$
**Result**: a set of feature vectors F corresponding to the instances I
**Comment:** $pathMap$ 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 undifned
            - set $pathMap(path) = pathIdx$ - set$pathIdx = pathIdx + 1$
        - set $fv(pathMap(path)) = fv(pathMap(path)) + 1$
        - do $processVertex(o, root, path, fv, d - 1)$
```

This algorithm can be directly applied to small and medium size datasets. However, big datasets are commonly distributed on different nodes as a *Resilient Distributed Dataset* (RDD), or *DataFrame* (DF) [6] and should be processed in parallel by using frameworks like Spark and SANSA. One main aspect of the ITP kernel is the iterative construction of paths and the associated $pathMap$ that assigns a unique integer to each path. To optimize the performance on distributed data we want to avoid this iterative and not parallelized construction if possible.

This can be achieved by using a different representation of the constructed trees as shown in algorithm 2. To keep track of the different *Uniform Resource Identifiers* (URIs) of subjects and objects during the parallelized construction of trees, we start off by mapping these URIs iteratively to integers. Furthermore we map the instances to their respective label and transform the TripleRDD, generated by using the SANSA RDF utilities, to a DataFrame. Construction of the paths of trees is implemented as a series of SQL queries on DataFrames utilizing the inherent structure. Instead of representing each path as an integer on a $pathMap$, as in the ITP kernel, we construct each path in *pathDF* as a *String* containing the predicates and objects respectively subjects.

Based on all the paths created and stored in *pathDF* we can now compute the trees by aggregating paths and collecting them as an Array[*String*] for each subject. Now those Array[*String*] can not only be interpreted as a tree in form of a list of paths, but also as a regular text document using paths as a vocabulary. In doing so we can make use of the *Spark ML CountVectorizer* that is designed to convert a collection of text documents to vectors of token counts, extracting the vocabulary in form of a sparse representation. This basically replaces the $pathMap$ with the huge upside of performing parallelized on distributed data.

The Array[*String*] of each subject is transformed to a sparse vector containing information about which paths are part of the tree with a consistent mapping for all subjects covering all the occurring paths. These sparse vectors can be used as feature vectors for different machine learning algorithms of the Spark ML/MLlib libraries like logistic regression or random forests.

Algorithm 2: The RDFFastTreeGraphKernel

```
Data: a TripleRDD $R$, an instance DF $I$ and a max depth $d_{max}$
Result: a set of feature vectors F corresponding to the instances I

Initialization:
    - map subjects and objects in $R$ and $I$ to unique integers
    - map instances in $I$ to their label
    - transform $R$ into a DF $T$ with columns $(s, p, o)$ for subject, predicate and object

Construct the paths of trees using SQL-Queries:
    - create a new $pathDF$ with columns $(s, path, o)$, where $path$ is the concatenation of
    $p$ and $o$ based on $T$
    - for depth $d$ in $[2, d_{max}]$:
        - create an empty $DF_d$ with columns $(s, path, o)$
        - find rows $r_s = (s_s, path_s, o_s)$ in the $pathDF$ where $o_s$ is the subject $s_o$ of
        another row $r_o = (s_o, path_o, o_o)$
        - add these rows to $DF_d$ as $(s, path, o) = (s_s, path_s + path_o, o_o)$
    - add all the $DF_d$ to the $pathDF$

Construct the feature vectors:
    - drop the column $o$
    - aggregate the rows of $pathDF$ for each $s$ and collect all the $path$ as a list in a new
    column $paths$ of type Array[$String$]
    - use the Spark ML CountVectorizer to transform this set of Array[$String$] to sparse
    feature vectors per subject
```

Based on our first experiments with the RDFFastTreeGraphKernel (see Section 4) and the results in [2] we want to further optimize the algorithm. In fact we do not always need to construct trees with a high depth d_{max}, and can instead use trees with $d_{max} = 1$ without loosing too much prediction accuracy. This allows us to simplify the Initialization massively by skipping the iterative and not properly optimized Scala-mappings required for tree construction and instead only operate on a slightly modified DataFrame which results in a simplified RDFFastGraphKernel as shown in algorithm 3.

In contrast to the RDFFastTreeGraphKernel that is still using some non parallelized Scala utilities, the RDFFastGraphKernel is fully parallelized and only utilizing Spark functions for RDD and DataFrames. This is reducing the computation time massively and allows an efficient computation of feature vectors for big datasets. Since we are only operating with $d_{max} = 1$, those feature vectors only contain information about the direct surroundings of instances. In some cases this information may not be enough for classification algorithms, however our tests show that the dimensionality reduction of feature vectors associated with this loss of information can also be beneficial for classification.

Algorithm 3: The RDFFastGraphKernel

```
Data: a TripleRDD $R$, a String predicateToPredict
Result: a set of feature vectors F corresponding to the instances of R

Construct the paths and assign classes
    - create a new pathDF with columns $(instance, class, path)$ by mapping $R$ to a DF
        - if a row $(s, p, o)$ in $R$ contains the predicateToPredict we map to $(s, o, "")$
        - else we map the row $(s, p, o)$ to $(s, "", p + o)$ and store the path

Construct the feature vectors:
    - aggregate the rows of pathDF for each $s$, set class to the only non-empty entry,
    and collect all the path as a list in a new column paths of type Array[String]
    - use the Spark ML StringIndexer to transform the set of classes to numerical labels
    - use the Spark ML CountVectorizer to transform this set of Array[String] to sparse
    feature vectors per subject
```

3 Implementation

We implemented the RDFFastTreeKernel as a component of the SANSA-ML library of SANSA-Stack, so that SANSA-Stack users can apply RDFFastTreeKernel to their ML workflow. Our focus was not only implementing the kernel as proposed in the paper [2], but also keeping the computational benefits of distributed operations. For that reason, we have tried several different implementations and ultimately three different kernel implementations have remained.

Source Location:
SANSA-ML/sansa-ml-spark/src/main/scala/net.sansa_stack.ml.spark/kernel/

Package:
Net.sansa_stack.ml.spark.kernel

3.1 Class Inputs

1. RDFFastGraphKernel

sparkSession: *SparkSession*	Spark session should be passed
tripleRDD: *TripleRDD*	Loaded *TripleRDD* object
predicateToPredict: *String*	Target predicate of link-prediction task

2. RDFFastTreeGraphKernel

sparkSession: *SparkSession*	Spark session should be passed
tripleRDD: *TripleRDD*	Loaded *TripleRDD* object in which the prediction target related triples are excluded
instanceDF: *DataFrame*	Known instances and labels (classes)
maxDepth: *Int*	Maximum depth of subtrees

3. RDFFastTreeGraphKernel_v2

sparkSession: *SparkSession*	Spark session should be passed
tripleRDD: *TripleRDD*	Loaded *TripleRDD* object in which the prediction target related triples are excluded
instanceDF: *DataFrame*	*DataFrame* that contains instances and labels from the loaded triples
maxDepth: *Int*	Maximum depth of subtrees

3.2 Class Functions

Since all three classes/objects are based on the same kernel approach, they do have the same class functions.

Function	Input	Output
computeFeatures	-	DataFrame root |– instance: integer |– paths: array | |– element: string |– label: double |– features: vector
getMLFeatureVectors	-	DataFrame root |– label: double |– features: vector
getMLLibLabeledPoints	-	RDD[LabeledPoint]

computeFeatures: Construction of linked paths from each instance and computation of feature vector is done in this function.
getMLFeatureVectors: Wrapper of computeFeatures, which provides data in structure for DataFrame-based Spark ML models.
getMLLibLabeledPoints: Wrapper of computeFeatures, which provides data in structure for RDD-based Spark MLlib models.

4 Evaluation

We have tested this algorithm on the datasets used in the experiments of [2]. This includes the affiliation prediction experiment in which we predict the affiliations for people in the AIFB research institute based on the data from their semantic portal. Also tested was the multicontract prediction using data from the linked data data-mining challenge 2013[3]. Here the task is to predict whether a contract has the property that it is a multicontract. As a final prediction experiment we try to replicate the geological theme prediction experiment using the named rock units in the British Geological Survey[4] data.

[3] http://keg.vse.cz/dmold2013/data-description.html
[4] http://data.bgs.ac.uk/

To set up the experiment environment, we used LogisticRegressionWithLBFGS of Spark MLlib to fit a logistic regression model for multiclass classification with 10-cross-fold validation. We measured average accuracy and runtime on each task using each kernel.

Table 1: Affiliation prediction experiment with 1 iteration per data set

		$d_{max} = 1$	$d_{max} = 2$	$d_{max} = 3$
Fast Tree Graph	Accuracy	0.846	0.866	0.75
	Time in s	40.296	51.041	1544.262
Fast Tree Graph v2	Accuracy	0.761	0.857	
	Time in s	37.159	113.314	

4.1 Prediction accuracy

One important aspect of the performance is the accuracy regarding the prediction experiments. However, the direct comparison of accuracies between the result of our experiments and the result from [2] is not applicable, since we are using LogisticRegressionWithLBFGS of Spark MLlib while [2] is using the C-SVC support vector machine algorithm from LibSVM. Furthermore there currently exists no other implementation of different RDF graph kernel algorithms in Spark yet, which means that a comparison between competing kernels is not applicable. Instead we try to show that our algorithms are capable of reaching a level of prediction accuracy similar to what has been shown in [2]. The full results for $d_{max} = 1$ are shown in table 2.

As you can see there are no significant differences between the different algorithms. Only in the affiliation prediction the FastTreeGraph performed better than the other versions, which should be studied further. In table 1 we also included a test with larger values of d_{max}. While computation with $d_{max} = 2$ resulted in higher accuracies and still decent computation times, the results for $d_{max} = 3$ were not worth the effort. In fact these large trees lead to problems, causing the learning on the feature vectors of FastTreeGraph to be significantly slower. FastTreeGraph_v2 did not even finish properly, which also requires further studies.

4.2 Runtime

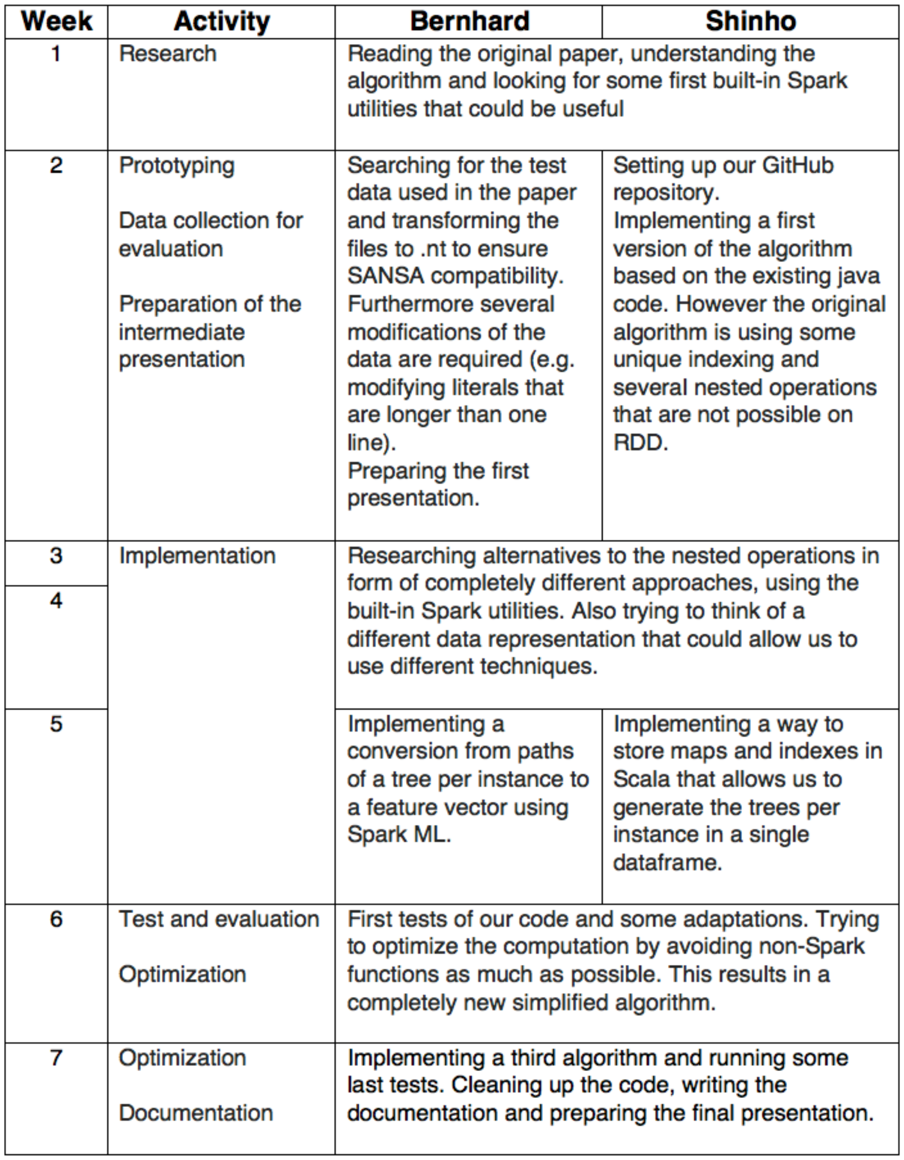
To test the runtimes of the different kernel implementations on different dataset, we measured timestamps after each initialization, feature vector computation and the learning model. Table 2 shows the runtime for each kernel, task and step with $d_{max} = 1$. Note that the runtime in the intermediate step does not include the full computations since RDD and DataFrame operations are lazy. The real computation is conducted when the data is used.

However it is clearly visible that FastGraph scales nicely, being about twice as fast as our FastTreeGraph, even for the large Theme prediction datasets. Furthermore we expect this performance gap to increase even more on larger datasets using a cluster with multiple nodes, since this algorithm is only operating on the distributed *Spark* data. In case of FastGraph and FastTreeGraph_v2, most of the time is spent on the LogisticRegressionWithLBFGS, even considering the lazy behaviour of DataFrame operations and our way of measuring the times.

Table 2: Test results with $d_{max} = 1$ and 10 iterations per data set measured in seconds

		Affiliation	Multi contract	Theme 10%	Theme 20%	Theme 40%	Theme 60%	Theme 80%	Theme 100%
	Num of triples	28,429	141,976	31,390	62,779	125,546	188,307	251,061	313,813
	Num of Instances	177	208	1,142	2,269	4,543	6,834	9,115	11,397
Fast Graph	Initialization	14.447	19.912	17.183	17.249	10.066	13.170	16.310	21.440
	FV Comp/Read	0.679	1.017	1.499	2.423	2.800	5.959	10.580	17.879
	Learning/testing	133.857	124.369	160.868	224.333	488.627	1278.625	2275.036	3405.255
	Accuracy	0.788	0.803	0.869	0.912	0.932	0.905	0.925	0.926
Fast Tree Graph	Initialization	22.385	3024.433	32.373	167.171	939.274	2231.309	4529.312	8592.241
	FV Comp/Read	0.710	0.864	1.237	1.634	1.905	2.309	3.070	3.747
	Learning/testing	81.539	81.523	100.736	100.686	195.152	287.003	364.581	455.612
	Accuracy	0.867	0.782	0.896	0.904	0.933	0.909	0.912	0.921
Fast Tree Graph v2	Initialization	16.924	53.996	23.736	33.304	51.203	70.979	97.742	122.563
	FV Comp/Read	1.435	1.824	2.429	4.526	8.289	15.425	25.380	38.786
	Learning/testing	108.716	154.014	225.941	337.245	876.925	2095.901	2917.838	4918.318
	Accuracy	0.761	0.788	0.888	0.906	0.931	0.901	0.917	0.925

5 Project timeline



Week	Activity	Bernhard	Shinho
1	Research	Reading the original paper, understanding the algorithm and looking for some first built-in Spark utilities that could be useful	
2	Prototyping Data collection for evaluation Preparation of the intermediate presentation	Searching for the test data used in the paper and transforming the files to .nt to ensure SANSA compatibility. Furthermore several modifications of the data are required (e.g. modifying literals that are longer than one line). Preparing the first presentation.	Setting up our GitHub repository. Implementing a first version of the algorithm based on the existing java code. However the original algorithm is using some unique indexing and several nested operations that are not possible on RDD.
3	Implementation	Researching alternatives to the nested operations in form of completely different approaches, using the built-in Spark utilities. Also trying to think of a different data representation that could allow us to use different techniques.	
4			
5		Implementing a conversion from paths of a tree per instance to a feature vector using Spark ML.	Implementing a way to store maps and indexes in Scala that allows us to generate the trees per instance in a single dataframe.
6	Test and evaluation Optimization	First tests of our code and some adaptations. Trying to optimize the computation by avoiding non-Spark functions as much as possible. This results in a completely new simplified algorithm.	
7	Optimization Documentation	Implementing a third algorithm and running some last tests. Cleaning up the code, writing the documentation and preparing the final presentation.	

Further ideas:
Up until now the algorithms have only been tested locally with medium size data files. The next step would be tests with bigger data on a real cluster with multiple nodes to see how the different algorithms perform under those conditions. While the basic RDFFastTreeGraphKernel will probably struggle with long initialization times, the other two alternatives should be quite performant. Furthermore it would be interesting to see how different learning algorithms like SVM or neural networks deal with the feature vectors generated by our kernels. Maybe the sparsity of our feature vectors can be further exploited.

References

[1] Gerben K. D. de Vries. *A Fast Approximation of the Weisfeiler-Lehman Graph Kernel for RDF Data*, pages 606–621. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[2] Gerben Klaas Dirk De Vries and Steven De Rooij. A fast and simple graph kernel for rdf. In *Proceedings of the 2013 International Conference on Data Mining on Linked Data - Volume 1082*, DMoLD'13, pages 23–34, Aachen, Germany, Germany, 2013. CEUR-WS.org.

[3] Yi Huang, Volker Tresp, Markus Bundschus, Achim Rettinger, and Hans-Peter Kriegel. *Multivariate Prediction for Learning on the Semantic Web*, pages 92–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[4] Achim Rettinger, Matthias Nickles, and Volker Tresp. *Statistical Relational Learning with Formal Ontologies*, pages 286–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[5] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, New York, NY, USA, 2004.

[6] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.