An implementation of RDF-Graph Kernels for Spark

Dennis Kubitza *

Maksym Radomskyi †

February 22, 2018

Project References on Github

Lab Report [1]

Abstract

Many implemented machine learning algorithms strongly depend on the specific structure of the observed and unobserved data, which forces users to fit their observations in a particular predefined setting or re-implement the algorithms to fit their requirements. For dynamic data models like Resouce Description Frameworks, that operate on schema-free structures, one class of algorithms is naturally well-suited to compromise both approaches: kernel based algorithms. We follow the research of Lösch et al. (2012) and implement graph kernels described in the paper for the usage with Apache Spark, especially for further usage in the Semantic Analytics Stack (SANSA). Our implementation combines different approaches from graph combinatorics, data mining and big data analysis to ensure scalability in storage and computational performance.

*Email: **denn_kubi@freenet.de**, Rudolf-Breitscheid-Str.1 40595 Düsseldorf, Germany

†Email: **maxradomskyy@gmail.com,**

[1] as Part of the Examination of Modul 4223, Master of Computer Science, University of Bonn

Contents

1 Introduction

While each and every machine learning task is defined by its input set, its set of valid models and the expected behavior of the learning agent, some algorithms exist that solve problems under such general assumptions that almost any data-dependent problem can be reduced to fit their requirements. Kernel-based machine learning methods don't require any specific structure for the data, only the existence of a scalar valued function, suitable for summarizing an observation or sub-observation as a single value. We call these functions kernel functions. Such algorithms are of especially high value for schema-free data like RDFs or labeled property graphs, as they neither enforce to rearrange the data or rewrite existing algorithms and paradigms. As kernels only need to fulfill very basic properties, kernel-based machine learning approach is very flexible in the definition of a learning task. In particular for knowledge graphs like RDFs, where we can implement different local and global models in a rather unsophisticated way. In the context of growing databases, usage of kernels for different tasks also offers new possibilities such as calculation of the latent feature models, which are commonly used to analyze global relations and can get rather costly, especially due to machine learning approaches needed to train them. In this lab we will try to give efficient implementations for the computation of some basic graph kernels, described in Lösch et al. (2012), back-boned by the Spark environment. Although our main source of information describes some unsuitable or faulty methods, we manage to provide the implementation of all described kernels. This report is structured as follows: in following section Theory and Approach we will define the kernels in the theory, and give the idea behind the described graph kernels Lösch et al. (2012). For each of the described kernels we will state the problems that we were facing, discuss issues with the suggestions of Lösch et al. (2012) and our solution proposals. In the section Implementation we will then describe the most interesting parts of the concrete implementation of the final version of the package and the used structures. The final part concerning the implementation will be Evaluation, where we describe our test procedure and the computation time performance of different solution attempts. In the end we will provide a final statement concerning the work flow we set up and the major difficulties we had to cope with during the implementation.

2 Theory and Approach

A definition of kernels, that is suitably general and applicable to all machine learning algorithms, but still mathematically precise can be found in (Shawe-Taylor and Cristianini, 2004). In the context of RDF graphs it is possible to reformulate this definition, as Lösch et al. (2012) did, bridging machine learning to feature-representation models for structured Data.

Definition 1 *Let $\mathscr{X}$ be a subgraph of an RDF graph and let $\phi : \mathscr{X} \Rightarrow \mathbb{R}^k$ be a feature representation. A kernel function is given by*

$$k(x, y) =< \phi(x), \phi(y) >_H$$

, where $< \cdot, \cdot >_H$ extends $\mathbb{R}^k$ to a Hilbert space.

In the case of RDF data, Lösch et al. (2012) listed four different kernels that are particularly suitable, as they may scale to both local and global features and compute kernels independently from the type of reference or literals given. This is due to the used feature representation, where the existence of certain characteristic subgraphs are identified with indicator variables in the feature representation of two subgraphs of interest. In the following paragraphs we will explain them and discuss possible problems / computation approaches, we want to consider in our implementations.

2.1 Walk Kernel

The walk kernel corresponds to a weighted sum of the cardinality of walks up to a length l, or more formally:

$$(\kappa_{l,\lambda})(G_1, G_2) = \sum_{i=1}^{l} \lambda^i \big|\{p | p \in walks_i(G_1 \cap G_2)\}\big|$$

Lösch et al. (2012) proposed to calculate the number of paths either by breadth-first search or by multiplication of the adjacency matrix. Although we decided to implement the first approach, we also added the method using the adjacency matrix for comparison, as we already implemented the necessary sub-routines. Both implementations are build on Spark's libraries GraphX and MLLib, as they provide a suitable parallelization of the necessary routines. While the matrix approach is self explaining, we were in need of a fast distributed algorithm for the graph approach. Consider that we know how many paths of length i end in each node, than we can construct the paths of length i+1, by summing up the the paths of length i from each ingoing neighbor. To implement such an algorithm, it is necessary to send information over our partitioned graph, most favorably in a parallelized way. The GraphX library contains a function called aggregateMessages() which implements exactly this feature and is, therefore the backbone of our function. For further details we refer to **??**, where we provide the most interesting code snippets.

2.2 Path Kernel

The path kernel uses the same principle as the walk kernel, but counts the number of paths instead. Unlike walks, path are not allowed to contain repeating elements, i.e. must consist of distinct vertexes.

$$(\kappa_{l,\lambda})(G_1, G_2) = \sum_{i=1}^{l} \lambda^i \big|\{p | p \in paths_i(G_1 \cap G_2)\}\big|$$

As it is now not possible to use the path independence, as before, we also need to alter the approach, most favorable without the need of accessing previous calculations steps, e.g to check if we already visited a single node. Lösch et al. (2012) suggested to use following formula for calculating the paths of length up to l:

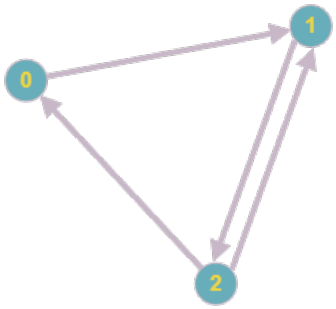
$$|paths_l| = \sum_{k=1}^{i} \sum_{j=1}^{m} \sum_{k=1}^{m} (M^i)'_{(j,k)}$$

where $(M^i)'$ is the i-th potency of the adjacency matrix with diagonal elements replaced recursively by 0. Unfortunately this method does not provide the exact cardinality but only an upper bound (counterexample: 1). We therefore added a second method for path computation, also relying on aggregateMessages(). In difference to the approach in 2.1 we had to extend the messages from integers to a sets of lists resembling all paths that led to a node. In each iteration step a receiving node will collect all paths and rule them out if its own *id* is contained.

2.3 Subtree Kernel

As Lösch et al. (2012) already mentioned the computation of a intersection graph might get costly, especially when the data is not distributed with intelligent indexing. They therefore proposed to limit the calculations of kernels, not on arbitrary subgraphs, but only on certain subgraphs which can be identified with a certain central entity, and share a common structure. This enables a replacement of the intersection graph with other suitable structures. One of them is the so-called intersection tree of depth d: $T(G, e_1, e_2, d)$:

Figure 1: Counterexample



$$(M^1)' = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad (M^2)' = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad (M^3)' = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

(Graphics generated with graphonline.ru)

Definition 2 *$T(G, e_1, e_2, d)$ is the result of following modifications:*

- *Intersect the Instance Trees (Lösch et al. (2012)) of depth d*
- *Replace each occurrence of e_1, e_2 with a new Label*
- *Prune everything that is not reachable from e_1, e_2*

For the full algorithm we reference the paper Lösch et al. (2012). Unfortunately, this setting is not suitable for the data structures provided by Spark. There are two general approaches that we would suggest for computation: either we use a local search on all nodes in a parallel manner, like in 2.1 or directly iterate for each node through its neighbors. First approach would be inefficient, as all components of the graph have to be evaluated, the second one would yield prohibitive lookup-times in the distributed RDDs.

As both full subtree and partial subtree kernels, suggested by Lösch et al. (2012), rely on the number of structures contained in the intersection tree, an efficient algorithm for computing the intersection tree is nevertheless necessary. We decided again to use an approach with GraphX to generate the intersection tree in, at least, acceptable time. Instead of the proposed Algorithm 1 Lösch et al. (2012) we parallelized all iterations again by aggregateMessages() to build the desired directed tree simultaneously from all common neighbors of e_1 and e_2. To ensure that the tree property holds, we generate a new id at random for each node at each iteration step.

The final kernels can now be defined by following functions operating on the intersection tree. Their names hereby originate on what is actually counted and how it is weighted:

2.3.1 Full Subtree Kernel

The full subtree kernels are based on the number of full subtrees contained in the intersection graph. A full subtree of tree t starting at some node $v \in t$ is a tree, whose root is v and which contains all descendants of v. We perform calculation of the number of full subtrees, which corresponds to the number of nodes with descendants, and apply some weight on the height of each obtained subtree.

Definition 3

$$\kappa_{FT}(e_1, e_2) = st(root(itd(e_1, e_2))), \quad with \quad st(v) = 1 + \lambda \sum_{c \in children(v)} st(c).$$

As the result of our intersection tree algorithm is again a graph object, the calculation of the corresponding value can easily be distributed. Suppose that we have the intersection tree - starting

by the leaves, we can send the current iteration value up to its ancestor. If it received messages from all its descendants it can calculate its own value and forward it.

2.3.2 Partial Subtree Kernel

Next step is calculation of partial subtree kernels. Partial subtrees don't require that all children of each node are contained in it, just any positive amount of them. Again they are weighted based on their heights. To avoid the calculation of each single subtree and weighting it, the following equivalent formula can be used (Lösch et al., 2012) :

Definition 4

$$\kappa_{PT}(e_1, e_2) = t(root(itd(e_1, e_2))), \quad with \quad t(v) = \prod_{c \in children(v)} (\lambda t(c) + 1).$$

The implementation approach is hereby the same as in 3, but with a different function for agglomeration.

3 Implementation

As our code grew bigger and more complex than we expected, we will only present the function headers to give a short overview, together with an explanation of the final method and structure. For a detailed version, we refer to our commented code, which can be automatically formated to a ScalaDoc. Unfortunately we have to admit, that we couldn't implement the tree kernel functions in time, as the intersection tree algorithm needed some adaption in the last minute. Our final package is called rdfKernels and provides the class KernelFunctions, containing all relevant methods.

```
package rdfKernels
[...]
class KernelFunctions {
[...]
}
```

3.1 Matrix based Kernels

One of the approaches to kernels computation that we implemented is based on matrices. As already discussed, values of graph kernels are dependent on the amount of possible paths and walks. Adjacency matrix, being a convenient way to calculate number of former and suitable enough for approximating number of latter (to the extent which is required by kernels computation) is the obvious choice. We start with calculating input graphs intersection and then create a square matrix, each element M_{ij} of which is set to '1' if there exists a connection between vertices i and j and '0' otherwise.

```
def intersectionAdjacencyMatrix(firstRDD: RDD[Triple],
        secondRDD: RDD[Triple]) : CoordinateMatrix ={
   [...]
   return adjacencyMatrix
}
```

In order to compute paths and walks of length more than 1, we need to elevate our adjacency matrix to the power which equals desired length. Since our matrices are instances of CoordinateMatrix, there exists no native method for their direct multiplication. Thus, we implemented a rather naive, nevertheless efficient enough method for this operation.

```
def coordinateMatrixMultiply(leftMatrix: CoordinateMatrix,
        rightMatrix: CoordinateMatrix): CoordinateMatrix = {
    [...]
return new CoordinateMatrix(productEntries,
leftMatrix.numRows(), rightMatrix.numCols())
}
```

Having these two functions, computing path kernels is rather trivial - we use the formula, described in the paper and produce a weighted sum of all non-zero entries, with each iteration increasing power of adjacency matrix by 1 until the required depth is met.

```
def matrixPathKernel(adjacencyMatrix: CoordinateMatrix, depth:
        Int, lambda: Double) : Double = {
    [...]
     return pathKernel
}
```

Walk kernels are obtained in the same manner, we just need to set all diagonal entries M_{ii} to zero which, as claimed by paper authors should give us suitable results for kernel computations.

```
def matrixWalkKernel(adjacencyMatrix: CoordinateMatrix,
depth: Int, lambda: Double) : Double = {
        [...]
    return walkKernel
}
```

3.2 Graph based Kernel

Next two implemented approaches for kernel computation rely on sending messages over the graph. Both functions take two instances of RDD[Triple] as inputs, the depth d and the scaling factor. They both rely on the alternating method of sending messages, agglomerating them, and merging the new values in the Graph with each iteration.

For the Walk Kernel the iterative Messages contain the current number of walks of length i ending in a Sender. A receiver sums up all messages and generates the count of walks of length i+1. To calculate the number of all paths, this information is collected and again summed up.

```
def msgWalkKernel(firstRDD: RDD[Triple], secondRDD: RDD[Triple],
        depth: Int, lambda: Double) : Double = {
```

```
    [...]
    return kernel
}
```

The Path kernel extends the previous idea, by not only sending the number of paths, but also for each and every one of them its whole history. After each node send his message for iteration step i, each node will check if its ID is already contained in the path and if so rule them out.

```
def msgPathKernel(firstRDD: RDD[Triple], secondRDD: RDD[Triple],
depth: Int, lambda: Double) : Double = {
    [...]
    return kernel
}
```

3.3 Intersection Tree

The intersection tree algorithm relies on the same principle of iterative message propagation as the msgPath Algorithm. Each node stores during the iteration the Paths of length i ending in it and collects the information of paths of length i+1. The final Intersection tree is constructed by using two different calls of helper functions addToRoot() and addToParent(), after each Node sent its current Set of Paths over all Outgoing Edges. To maintain the structure of the instance graph, instead of using original IDs, random IDs were generated at each iteration step for each incoming path. The main problem was to generate the Random IDs for portioned Data, as the random generators might synchronize. Therefore each Node Keeps track of a recursively generated Seed, from an initial global starting point. After each iteration, and after the creation of new Vertex IDs this value had to be updated.

```
def constructIntersectionTree(graphRDD: RDD[Triple],
        e1: Long, e2: Long, depth:Int, spark: SparkSession)
        : Graph[Null, String] = {
   [...]
   return Graph.fromEdges(edges, null)
}
```

3.4 Full and Partial Subtree Kernels

Both subtree algorithms operate on the tree structure generated by the intersection tree algorithm. They take this input and revert the tree. After that each former leaf, will receive an initial signal of 1 and send them in the direction of root along the tree structure. During the iterations each node counts the number of calls it received. If this number equals its original number of outgoing edges, it will apply the recursive formulas described in 2.3.1 2.3.2 and send a signal to the

```
def fullSubtreeKernel(g: Graph[Null,String],
        depth: Int, lambda: Double) : Double = {
```

```
        [...]
    return(kernel)
}

def partialSubtreeKernel(g: Graph[Null,String],
        depth: Int, lambda: Double) : Double = {
        [...]
    return(kernel)
}
```

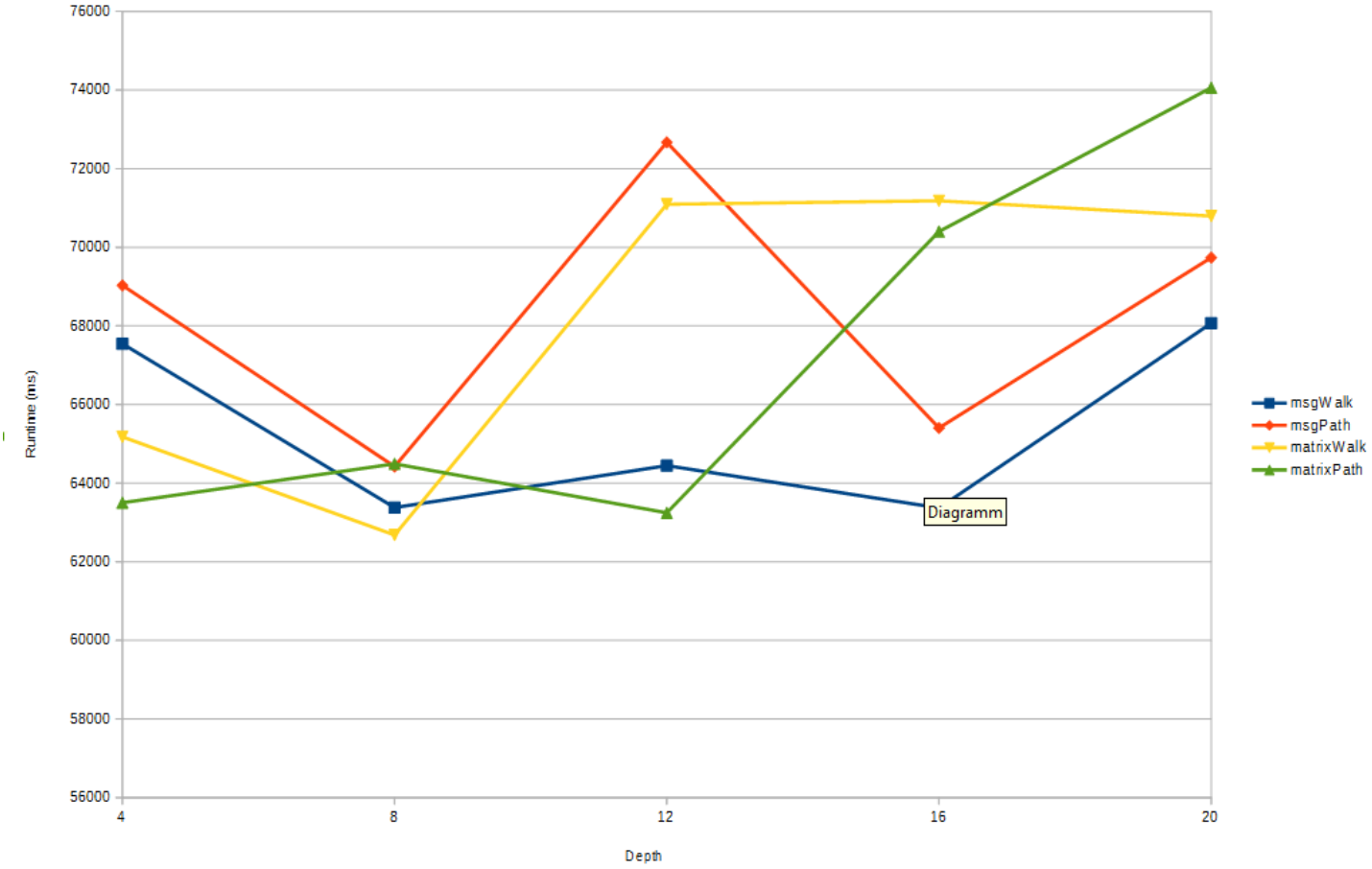
4 Evaluation

As we managed to provide different algorithms for the Path/Walk-Kernels and mentioned the theoretical problems with the computation of graph kernels, we intended to inter-compare them. We managed to validate the correctness of each algorithm on small constructed subsets in N-Triple syntax and performed performance comparison on data provided by the DBpedia project, mainly on a subset of size 100MB from the dataset Infoboxes. The first evaluation step was the comparison of the functions matrixPathKernel(), matrixWalkKernel(), msgWalkKernel() and msPathKernel(). We transformed the dataset into an RDD serving as input for both function arguments of each function, resulting in a redundant intersection step. We then ran evaluations for depth 4,8,12,16 and 20. The results can be found in 2, as a table and a scaled graph, with log-scaled x- Axis.

5 Workflow

At the beginning of the implementation phase, we decided to not separate the task strictly between us, but to work together in weekly evening meetings. Therefore the following Table will not denote who has implemented a task, but who did the major contributions to them.

Figure 2: Runtimes on small Datasets



Week	Maksym	Dennis	General
19.12			Discussing the paper
02.01			Still attempting to set up VM with working Software
09.01			Still attempting to set up VM with working Software
16.01	Draft for Structure	Upload working VM Initialize Github Create LaTex Template	
23.01	Testing of VM		Set up Package
30.01	Matrix Approach Paths	Graph Approach Walks Draft Chapter 2	Testing Algorithms Recherches on counting Paths
07.02	Matrix Approach Walks	Graph Approach Walks Draft Chapter 1	Draft Chapter 2 Discuss Tree-Kernels
13.02	Draft Chapter 4	Intersection Tree Algorithm (Faulty)	Testing
20.02	Finalize Package Re-factor Code Improve comment	Creating Figures/Plots Evaluation Fix Intersection Tree	Finalize Report

Considering everything in the end, it was a bit complicated to set up the Hadoop environment. Out of the box the program was not able to generate datanodes. We figured out that the recent Ubuntu Version 17.10 needed some extra configuration, to ensure that the Java library can be loaded. If there is any need of it, we have a working VM provided at GoogleDrive. While matrix and graph based kernels were manageable to finish in adequate time, construction of intersection tree occurred to be a tough task, especially with intention to build an approach suitable for distributed processing. Testing revealed, that our first approach for the intersection tree algorithm produced faulty results, due to unexpected behavior of the aggregateMessage() function. Nevertheless, we managed to finish this step and after several validations on different datasets proceeded to kernel calculations which, as expected, turned out to be noticeably more trivial.

References

Lösch, U., Bloehdorn, S., and Rettinger, A. (2012). Graph kernels for rdf data. In Simperl, E., Cimiano, P., Polleres, A., Corcho, O., and Presutti, V., editors, *The Semantic Web: Research and Applications*, pages 134–148. Springer Berlin Heidelberg, Berlin, Heidelberg.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel methods for pattern analysis*. Cambridge university press.