# An implementation of RDF-Graph Kernels for Spark

Dennis Kubitza [*]

Maximilian Radomsky [†]

February 20, 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 examinations of Lösch et al. (2012) and implement their proposed Graph-Kernels for the usage in 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, it's 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 a observation or subobservation as a single value. We call such a function a kernel Functions. Especially for schema-free data like RDFs or Labeled Property Graphs such algorithms are highly valuable, as they neither enforce to re-factor the data or rewrite existing algorithms and paradigms. As Kernels only need to fulfill very basic properties, Kernel Based Machine learning is very flexible in the definition of a learning task. Especially for Knowledge Graphs like RDFs we can implement different local and global Models in a rather easy way. In the context of growing Databases the usage of Kernels for different Tasks also offers new possibilities as the calculation of Latent Feature Models, which are commonly used to analyze global relations, can get rather costly, especially as Machine learning is needed to train them. In this Lab we will try to give efficient implementations for the computing some basic graph Kernels, described by 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 the of the described Kernels, and the most important subroutine for the remaining two. 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 KernelsLösch et al. (2012). As we target the implementation of all 4 Kernels, we will state for each of them the problems that we expected to occur and state also 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 considering the implementation and package 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 in particular 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 in our partitioned Graph, most favorable in a parallelized way. The GraphX library contains the function aggregateMessages() which exactly implements 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.

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

. 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:

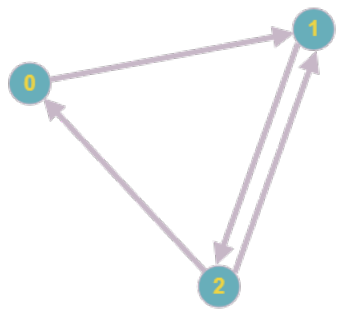
$$|paths_l| = \sum_{k=1}^{i} \sum_{j=1}^{m} \sum_{k=1}^{m} (M^i)'_{(j,k)}$$

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 whole Algorithm we reference to the paper Lösch et al. (2012).
Unfortunately this setting is not suitable for the Data-Structures provided by Spark. There are two general approaches, we would propose 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 kernels the Full-Subtree and Partial-Subtree kernel, proposed 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 an 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, in 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 is based on the Number of Full Subtrees contained in the Intersection graph. A Full Subtree is a Subgraph, for which every nodes descendants are also contained. Instead of calculating the number of full subtrees, which corresponds to the number of nodes with descendent's, also a weighing on the height of each subtree is applied.

**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 all messages from its descendent's it can calculate its own value and forward it.

### 2.3.2 Partial Subtree Kernel

Like the name suggest, we now consider and weight Partial Subtrees. Partial Subtrees don't require that all their ancestors are contained in it, just any positive amount of them. Again they are weighted by 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 longer and more complex than we expected, we will only present the function headers to give a short overview, together with a 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 kernels computations.

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

## 3.2 Graph based Kernel

As already mentioned both Graph based Kernel Methods rely on sending messages in the Graph. Both take two RDD Triples as inputs, the depth d and the scaling factor. Iteratively they both relay on the alternating method of sending messages, agglomerating them, and merging the new values in the Graph. As it was necessary to always save the State from each previous iteration, the methods grew very complex, especially the PathKernel.

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

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

### 3.3 The Intersection Tree

The Intersection Tree Algorithm relays on the same principle of iterative Sending Messages, and updating the values by a VertexJoin() and keeping track of the last states. While doing this to analyses the structure of the graph, we constructed the Intersection Tree by different Calls of Side-Functions AddToRoot(), AddToParent(). To remain the structure of an instance Graph, not the original Ids were used, but we regenerated a random ID for each Vertex and each iteration step, stored as a property.

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

## 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 validated the correctness of each algorithm on small constructed subsets in N-Triple Syntax on started to run the performance comparison on Data provided by the DBpedia Project, mainly on a subset of size 30MB from the Dataset Infoboxes. Unfortunetly we managed to only keep manual time stoppings. The functions matrixPathKernel(), matrixWalkKernel(), msgWalkKernel() and msPathKernel() where called with the subgraphs of distance d from a middle point, the functions construct IntersectionTree() with the corresponding entities. Please note, that we ran the instances on a single Cluster setup. The results on real Cluster might therefore differ. We obtained timings of 1:42, 1:49, 3:24 and 1:30 for the Path / Walk kernels and enormous 3:32 for the IntersectionTree().

## 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.

| Week | Maximilian | 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<br>Initialize Github<br>Create LaTex Template | |
| 23.01 | Testing of VM | | Set up Package |
| 30.01 | Matrix Approach Paths | Graph Approach Walks<br>Draft Chapter 2 | Testing Algorithms<br>Recherches on counting Paths |
| 07.02 | Matrix Approach Walks | Graph Approach Walks<br>Draft Chapter 1 | Draft Chapter 2<br>Discuss Tree-Kernels |
| 13.02 | Draft Chapter 4 | Intersection Tree Algorithm (Faulty) | Testing |
| 20.02 | Finalize Package<br>Re-factor Code<br>Improve comment | Creating Figures/Plots<br>Evaluation<br>Fix Intersection Tree | Finalize Report |

Considering everything in the end, it was far to 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. Despite of the initial issues the final testing revealed, that our first approach for the Intersection Tree Algorithm resulted in Faulty behavior, due to unexpected behavior of the aggregateMessage function. We tried our best to get everything running and succeeded, but lacked the time for finally implementing the Subtree Kernels. Nevertheless our package provides the functionality to generate the Intersection Tree as a Spark Graph structure. From that point on the implementation of both kernels should be feasible.

## 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.