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
TODO

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.

TODO: Add further information about implementation and/or results

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 Kernel (ITP) is given in Algorithm **??**.

[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 **??**. 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 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* and MLlib libraries like logistic regression or random forests.

TODO: Maybe add to evaluation and use only a reference instead. 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 multi-contract 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

---

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

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

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.

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

Class Input

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

Class Input

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

3.1 RDFFastGraphKernel

3.2 RDFFastTreeGraphKernel

3.3 RDFFastTreeGraphKernel_v2

3.4 Class functions

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

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.

Class Input

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

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]

4 Evaluation

5 Conclusion

5.1 Project timeline

5.2 Further ideas

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.