EasyChair Preprint

№ 478

RdfRules Preview: Towards an Analytics Engine for Rule Mining in RDF Knowledge Graphs

Václav Zeman, Tomáš Kliegr and Vojtěch Svátek

August 31, 2018

RdfRules Preview: Towards an Analytics Engine for Rule Mining in RDF Knowledge Graphs

Václav Zeman, Tomáš Kliegr, and Vojtěch Svátek

Department of Information and Knowledge Engineering,
Faculty of Informatics and Statistics, University of Economics Prague,
Czech Republic
{vaclav.zeman,tomas.kliegr,svatek}@vse.cz

Abstract. RdfRules is a framework for mining logical rules from RDF-style knowledge graphs. The system provides software support for the complete data mining workflows over RDF data: data ingestion, aggregation, transformations, actual rule mining and post-processing of discovered rules, including clustering. As a rule mining algorithm, RdfRules adopts AMIE+ (Galárraga et al, 2015), which has been extended with number of practical features, such as mining across multiple graphs, top-k approach and the ability to define fine-grained patterns to reduce the size of the search space. RdfRules is a work-in-progress.

Keywords: Rule Mining · RDF data analysis · Semantic Web tool · Knowledge Bases

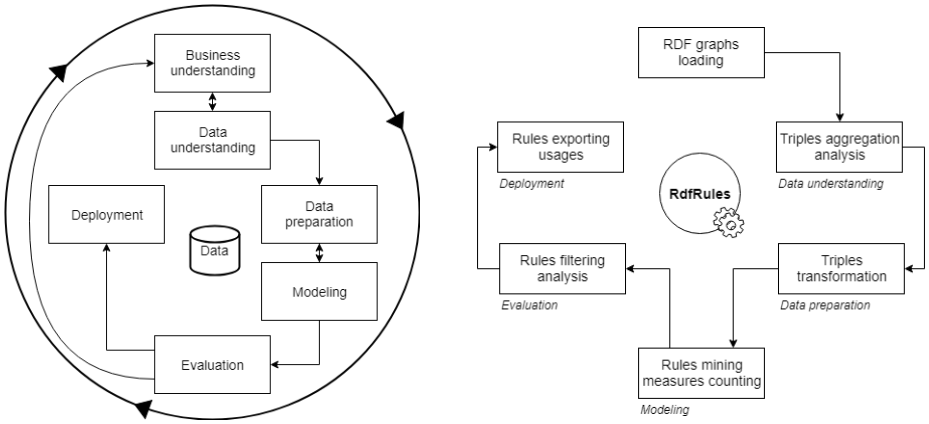
1 Introduction

Finding interesting interpretable patterns in data is a frequently performed task in modern data science workflow. Software for finding association rules, a specific form of patterns, is present in nearly all data mining software bundles. These implementations are based on the apriori algorithm or its successors, which are severely constrained with respect to the shape of analyzed data – only single tables or transactional data are accepted. Algorithms for logical rule mining developed within the scope of Inductive Logical Programming (ILP) do not have these restrictions, but they typically require negative examples and do not scale to larger knowledge bases [5].

Large knowledge bases consisting of linked and machine-readable data are currently typically published using the RDF[1] data representation [11]. RDF-style knowledge bases are sets of RDF statements which form labeled and oriented multi-graphs, and as such they do not contain negative examples. Each statement is written as a *triple* with subject-predicate-object or as a *quad* with additional information about a named graph attached to a given triple.

The current state-of-the-art approach for rule mining from RDF knowledge graphs is the AMIE+ algorithm [5]. Similarly to ILP systems, AMIE+ mines

[1] Resource Description Framework



(a) CRISP-DM processes (b) RdfRules processes

Fig. 1: The relationship between CRISP-DM and RdfRules processes.

Horn rules which have the form of implication and consist of one atomic formula (or simply *atom*) on the right side and conjunction of atoms on the left side.

$$hasChild(a, c) \wedge hasChild(b, c) \Rightarrow isMarriedTo(a, b)$$

The atom has just one specified predicate and two variables at the subject and object positions. One of these variables can also be replaced by some specific constant, e.g., $hasChild(a, Carl)$.

In this paper we describe the RdfRules framework, which uses AMIE+ as a basis for a complete solution for linked data mining. RdfRules adds pre-processing and post-processing capabilities, such as discretization of numerical attributes and clustering of the output rules. Furthermore, RdfRules provides several extensions over AMIE+, such as mining across multiple graphs, rule patterns, constraints, additional measures, top-k approach etc. The framework offers several ways to control mining processes either through a Scala and Java API, or through a REST web service with a graphical user interface.

This paper is organized as follows. Section 2 provides an overview of the architecture. Section 3 gives details on the implementation. A use case demonstrating a practical application of the system is presented in Section 4. Similar frameworks and other approaches are mentioned in Section 5. The conclusions summarizes the contribution and provides an outlook for future work.

2 Design and Architecture

An overview of data mining processes as implemented in RdfRules is shown in Fig. 1b.

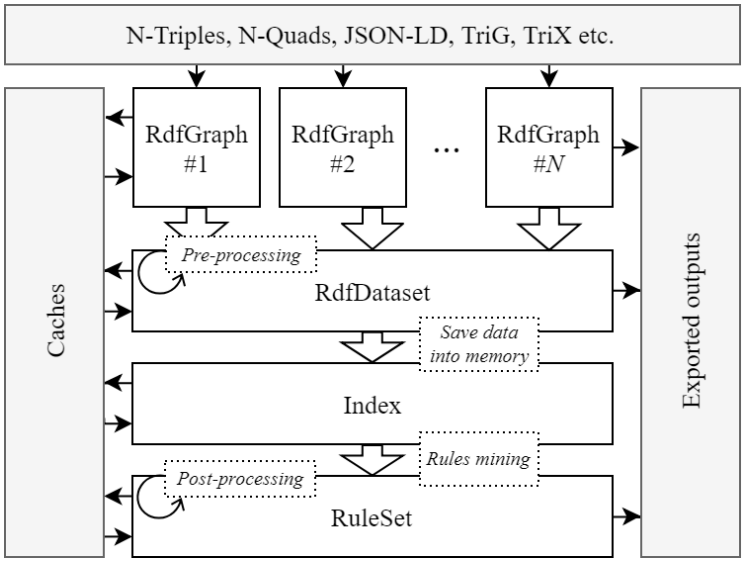


Fig. 2: Main data abstractions and processes in the RdfRules framework

2.1 Design paradigm

The construction and the sequence of individual processes have been inspired by the CRISP-DM methodology [10], which is depicted in Fig. 1a. The way the key CRISP-DM steps are supported in RdfRules is outlined below.

Data understanding and preparation. As the first step, RdfRules loads the input RDF knowledge graph. The framework offers several functionalities for data aggregation and analysis, such as computation of statistics on triples and their types. Informed by the performed analysis, in this stage, the user can also define transformations that are carried on the input RDF data.

Data modeling. In the modeling phase, the user restricts the search space for AMIE+, by defining pruning thresholds, rule patterns and possibly other constraints. Subsequently, the rule mining process is started and a complete set of found rules is obtained.

Evaluation. A critical phase in rule mining is sifting through the discovered rules to find the true "nuggets", rules that are of interest to the user. To support this, RdfRules allows the user to filter, sort, select and export interesting extracted rules based on number of features, including measures of significance (such as support, confidence and lift). There is also the option to cluster rules by their similarities.

2.2 Architecture

As Fig. 2 shows the architecture of the RdfRules core is composed of four main data abstractions: *RdfGraph*, *RdfDataset*, *Index* and *RuleSet*. Instances of these

abstractions are gradually created during processing of RDF data and rule mining. Each abstraction consists of several operations which either *transform* the current object or perform some *action* to create an output. Hence, these operations are classified as transformations or actions.

Transformations. Any transformation is a lazy operation that converts the current data object to another. For example a transformation in the *RdfDataset* object creates either a new *RdfDataset* or an *Index* object.

Actions. An action operation applies all pre-defined transformations on the current and previous objects, and processes (transformed) input data to create a desired output such as rules, histograms, triples, statistics etc. Compared to transformations, actions may load data into memory and perform time-consuming operations.

Caching. If we use several action operations, e.g. with various input parameters, over the same data and a set of transformations, then all the defined transformations are performed repeatedly for each action. This is caused by lazy behavior of main data abstractions and the streaming process lacking memory of previous steps. These redundant and repeating calculations can be eliminated by caching of performed transformations. Each data abstraction has the *cache* method that can perform all defined transformations immediately and store the result either into memory or on a disk.

2.3 Graphs and Datasets

The *RdfGraph* object is built once we load an RDF graph. It can either be a file or a stream of triples or quads in a standard RDF format such as N-Triples, N-Quads, JSON-LD, TriG or TriX. Besides these standard formats the RdfRules framework has its own native binary format to save/cache all defined data objects and transformations on a disk for later and repeated use. During the data loading process just one *RdfGraph* instance is created with a default or specified name. If the input data format supports named graphs, several instances can be created.

Any *RdfGraph* instance can be used as a set of triples with multiple transformation operations supported on them. One can filter triples by a condition, replace selected resources or literals and merge numeric data by discretization algorithms. Transformed data may be exported to a file in one of the RDF formats. For analytical purposes, the user can aggregate statements and view statistics or meta information about the graph such as types of predicate ranges or histograms of triple items. The complete list of all important operations is shown in Table 1.

The *RdfDataset* instance is created from one or many *RdfGraph* instances. It is composed of quads, where all triples have additional provenance information attached that expresses to which graph they belong. In addition to some extensions the *RdfDataset* object supports the same operations as the *RdfGraph* abstraction (see Table 2).

Table 1: The *RdfGraph* data abstraction: all important operations

Transformations	
map(*func*)	Return a new *RdfGraph* object with mapped triples by a function *func*.
filter(*func*)	Return a new *RdfGraph* object with filtered triples by a function *func*.
take(n) drop(n) slice(*from*, *until*)	Return a new *RdfGraph* object with filtered triples by cutting the triple set.
discretize(*task*, *func*)	Return a new *RdfGraph* object with discretized numeric literals by a predefined *task*. It processes such triples which satisfy a function *func*.
Actions	
foreach(*func*)	Apply a function *func* for each triple.
histogram(s, p, o)	Return a map where keys are items and values are numbers of aggregated items. Parameters s, p, o represents booleans determining which triple items should be aggregated.
types()	Return a list of all predicates with their type ranges and frequencies.
cache(*target*)	Cache this *RdfGraph* object either into memory or into a file on a disk.
export(*target*, *format*)	Export this *RdfGraph* object into a file in some familiar RDF format.

Table 2: The *RdfDataset* data abstraction: selected operations

This data abstraction has the same operations as the *RdfGraph*. The only difference is that operations do not work with triples but with quads.	
Additional transformations	
addGraph(*graph*)	Return a new *RdfDataset* with added *graph*.
index(*mode*)	Create an *Index* object from this *RdfDataset* object.

2.4 Indexing

Before mining the input dataset has to be indexed into memory for the fast rules enumeration and measures counting. The AMIE+ algorithm uses six fact indexes that hold data in several hash tables. Hence, it is important to realize that the complete input data are replicated six times and then stored into memory before the mining phase. This index may have two modes: *preserved* and *in-use*. The *preserved* mode keeps data in memory until the existence of the index object, whereas the *in-use* mode loads data into memory only if the index is needed and is released after use.

The *Index* instance can be created from the *RdfDataset* object or loaded from cache. It contains prepared data and has operations for rule mining with the AMIE+ algorithm (see Table 3).

Table 3: The *Index* data abstraction: all important operations

Transformations	
toDataset()	Return *RdfDataset* object from this *Index* object.
Actions	
cache(*target*)	Serialize this *Index* object into a file on a disk.
mine(*task*)	Execute a rule mining *task* with thresholds, constraints and patterns, and return a *RuleSet* object.

Table 4: Overview of mining thresholds supported by RdfRules

MinHeadSize	Minimum number of triples matching rule head. It must be greater than *zero*.
MinHeadCoverage	Minimal head coverage. It must be greater than *zero* and less than or equal to *one*.
MaxRuleLength	Maximal length of a rule. It must be greater than *one*.
TopK	Maximum number of returned rules sorted by head coverage. It must be greater than *zero*.
Timeout	Maximum mining time in minutes.

2.5 Rule Mining - State Space Restrictions

The AMIE+ algorithm outputs all rules matching the specified minimum values of selected measures of significance. This can output many more rules than desirable for the user, who may be, for example, interested only in rules that contain a specific attribute, or may want to prefer to see only the first 100 rules, rather than wait for the state space to be exhaustively searched. RdfRules addresses these user requirements by extending AMIE+ with the possibility to define new types of restrictions: new thresholds, rule patterns and constraints. These are described in greater detail in the following.

Thresholds. Besides standard thresholds defined in AMIE+, such as *support* and *head coverage*, RdfRules also offers the *top-k* approach and a *timeout* threshold which determines a maximum mining time. All mining thresholds are listed in Table 4.

Rule Patterns. RdfRules allows the user to specify several rule patterns using a pre-defined grammar. All rules must match at least one pattern from the rule pattern list. Matching is performed during the mining phase and therefore the rules enumeration can be greatly sped up thanks to stricter pruning of the state space.

$$AnyConst(AnyVar, AnyVar) \Rightarrow livesIn(AnyVar, AnyVar) \quad \text{(a rule pattern)}$$
$$wasBornIn(a, b) \Rightarrow livesIn(a, b) \quad \text{(a matching rule)}$$

Table 5: *RuleSet* data abstraction: overview of essential operations

Transformations	
map(*func*)	Return a new *RuleSet* object with mapped rules by a function *func*.
filter(*func*)	Return a new *RuleSet* object with filtered rules by a function *func*.
take(n), drop(n), slice(*from*, *until*)	Return a new *RuleSet* object with filtered rules by cutting the rule set.
filterByPatterns (*patterns*)	Return a new *RuleSet* object with rules matching at least one of the input rule patterns.
sortBy(*measures*)	Return a new *RuleSet* object with sorted rules by selected measures of significance.
computeConfidence (*minConf*)	Return a new *RuleSet* object with the computed confidence measure for each rule that must be higher than the *minConf* value.
computePcaConfidence (*minPcaConf*)	Return a new *RuleSet* object with the computed PCA confidence measure for each rule that must be higher than the *minPcaConf* value.
computeLift(*minConf*)	Return a new *RuleSet* object with the computed lift measure for each rule.
makeClusters(*task*)	Return a new *RuleSet* object with clusters computed by a clustering task.
findSimilar(*rule*, n), findDissimilar(*rule*, n)	Return a new *RuleSet* object with top n rules, the selected rules will be the most similar (or dissimilar) ones from the input rule.
Actions	
foreach(*func*)	Apply a function *func* for each rule.
cache(*target*)	Cache this *RuleSet* object either into memory or into a file on a disk.
export(*target*, *format*)	Export this *RuleSet* object into a file in some selected output format.

Constraints. Finally, the last mining parameter specifies additional constraints and defines a way of mining. Here is a list of implemented constraints that can be used:

- *OnlyPredicates(x)*: rules must contain only predicates defined in the set x.
- *WithoutPredicates(x)*: rules must not contain predicates defined in the set x.
- *WithInstances*: enable to mine rules with constants at the subject or object position.
- *WithObjectInstances*: enable to mine rules with constants only at the object position.
- *WithoutDuplicitPredicates*: rules that contain one predicate more than once will be removed.

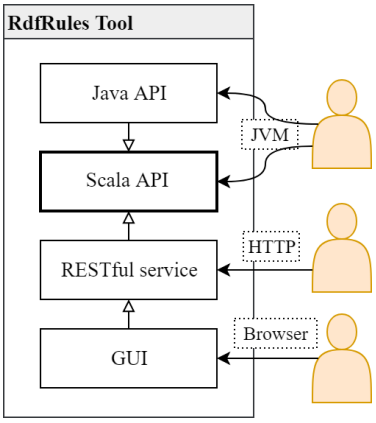


Fig. 3: Available interfaces in the RdfRules framework

2.6 Post-Processing of Discovered Rules

The *RuleSet* object is on the output of the RdfRules workflow. It contains all discovered rules conforming to the restrictions.

Every rule in the rule set consists of the head (consequent - the right side of the rule), body (antecedent - the left side of the rule) and measures[2] of significance. Basic measures are: *rule length*, *support*, *head size* and *head coverage*. Other measures may be calculated individually and explicitly within the *RuleSet* abstraction. Additional measures of significance include: *body size*, *confidence*, *PCA body size*, *PCA confidence*, *head confidence*, *lift* and *cluster*. Rules can be filtered and sorted by all these measures.

This final abstraction has multiple operations for rule analysis, counting additional measures of significance, rule filtering and sorting, rule clustering, and finally rule exporting for use in other systems (see Table 5). All the discovered rules are stored in memory but as in the case of previous data objects all transformations defined in the *RuleSet* are lazy. Therefore, this abstraction also allows to cache rules and transformations on a disk or in memory for a repeating usage. The complete rule set (or its subsets) can be exported and saved into a file in a human readable text format or in a machine readable JSON format.

3 Implementation

The core of RdfRules is written in the Scala language. Besides the Scala API, RdfRules also provides a Java API, REST web service and graphical user interface (GUI) via a web browser (see Fig. 3). The Scala or Java API can be used as a framework to extend another data mining system or application. The

[2] All measures of significance are described in the AMIE+ paper and on the RdfRules GitHub page: https://github.com/propi/rdfrules

web service is suitable for modular web-based applications and remote access. Finally, the GUI is based on the web service interface and can be used either as a standalone desktop application or as a web interface to control an RdfRules instance that is deployed on a remote server.

For RDF data processing RdfRules uses some modules from *Apache Jena*[3] framework. In the pre-processing phase, it is possible to use several discretization methods for automatic merging of numerical literals. These tasks represent facades for unsupervised discretization algorithms, such as equal-frequency and equal-width [3], implemented in the *EasyMiner-Discretization*[4] library, which is part of the EasyMiner system [9]. For post-processing the RdfRules uses cluster analysis to categorize the output rules by the *DBScan* algorithm [4].

The source code of RdfRules is published under the GPLv3 open-source license[5] and is hosted at GitHub[6]. Detailed manuals for using and deploying individual modules are described on the GitHub page.

4 Examples

Subsets of YAGO [8] and DBpedia [1] are used as example input datasets. These knowledge graphs are interconnected by the *owl:sameAs* predicate.

First, consider only the YAGO sample as the input knowledge graph. We can start the rule mining process simply by invoking several operations in the Scala API:

Scala API: rule mining with default parameters

```
Dataset("yago.tsv")
  .mine(Amie()) //defaults: MinHeadSize=100, MinHeadCoverage=0.01, MaxRuleLength=3
  .sorted       //sorted by HeadCoverage
  .take(3)      //take first 3 rules

//Output samples:
isMarriedTo(b, a) => isMarriedTo(a, b) | supp: 746, hc: 0.45, hs: 1667
participatedIn(b, c) ^ participatedIn(a, c) => dealsWith(a, b) | supp: 203, hc: 0.4, hs: 520
directed(a, b) => actedIn(a, b) | support: 60, headCoverage: 0.012, headSize: 4919
```

Notice that RdfRules automatically recognized the RDF format by the file extension. In this example the program first mines all rules conforming to all default restrictions. After that, it sorts rules by the head coverage and takes top three rules from the whole list.

The third output rule $directed(a, b) \Rightarrow actedIn(a, b)$ can be interpreted as follows: if someone directs something, e.g., a movie, then he or she is also acted in the movie. For this rule there are 4919 triples matching the head $actedIn(a, b)$, it is called *head size*. Only 60 triples of 4919 are connected to such triples that are matching the body $directed(a, b)$. In other words, there are only 60 cases of 4919 where an actor of a movie is also the director of the movie. This measure

[3] https://jena.apache.org/
[4] https://github.com/KIZI/EasyMiner-Discretization
[5] https://www.gnu.org/licenses/gpl.txt
[6] https://github.com/propi/rdfrules

is called *support*. A ratio between support and head size is called *head coverage*. For this example the head coverage is $\frac{60}{4919} = 0.012$.

This calculation may be sped-up by using the top-k approach, which returns the same results, but can be faster owing to gradual increase of the support threshold during mining.

Scala API: rule mining with top-k approach

```
Dataset("yago.tsv")
  .mine(Amie().addThreshold(Threshold.TopK(3)))
  .sorted
```

We can also compute additional measures of significance, such as *confidence*, *PCA confidence* (they indicate the quality of the rule) and *lift* (it indicates dependence between the body and the head). Similar output rules can be clustered by their similarity functions and the DBscan algorithm:

Scala API: rule mining with other measures

```
Dataset("yago.tsv")
  .mine(Amie()
  .addThreshold(Threshold.MinHeadSize(80))
  .addThreshold(Threshold.MinHeadCoverage(0.001))
  .addThreshold(Threshold.TopK(1000))
  .addConstraint(RuleConstraint.WithInstances(true))) //enable to mine rules with constants
  .computePcaConfidence(0.5)
  .computeLift()
  .makeClusters(DbScan())
  .sortBy(Measure.Cluster, Measure.PcaConfidence, Measure.Lift, Measure.HeadCoverage)
  .cache("rules-example3.cache") //save mined rules on disk

//Output sample:
participatedIn(a, <1999_NATO_bombing_of_Yugoslavia>) ^
participatedIn(b, <Attack_on_Aruba>) => dealsWith(a, b)
supp: 18, hc: 0.035, conf: 0.5, pcaConf: 0.56, lift: 78, cluster: 5
```

In the next example, we attach the DBpedia knowledge graph and trace such rules whose atoms belong to both graphs. This can be achieved by adding a rule pattern, which restricts the output rule set.

Scala API: rule mining across two graphs

```
(Dataset() + Graph("yago", "yago.tsv") + Graph("dbpedia", "dbpedia.ttl"))
  .mine(Amie()
  .addPattern(AtomPattern(graph = Uri("dbpedia")) =>: AtomPattern(graph = Uri("yago")))
  .addPattern(AtomPattern(graph = Uri("yago")) =>: AtomPattern(graph = Uri("dbpedia"))))
  .graphBasedRules //attach a graph to all atoms of all rules

//Output samples:
hasChild(a, c, <yago>) ^ parent(c, b, <dbpedia>) => isMarriedTo(a, b, <yago>)
hasChild(c, a, <yago>) ^ hasChild(c, b, <yago>) => relative(a, b, <dbpedia>)
hasNeighbor(a, c, <yago>) ^ spokenIn(b, c, <dbpedia>) => hasOfficialLanguage(a, b, <yago>)
```

All previous operations can also be performed using the Java API, web service and GUI. The following example shows a complex mining workflow in all available interfaces including pre-processing, rule mining, rules post-processing and exporting results to a file. Figure 4 shows a preview of defining individual processes in the GUI.

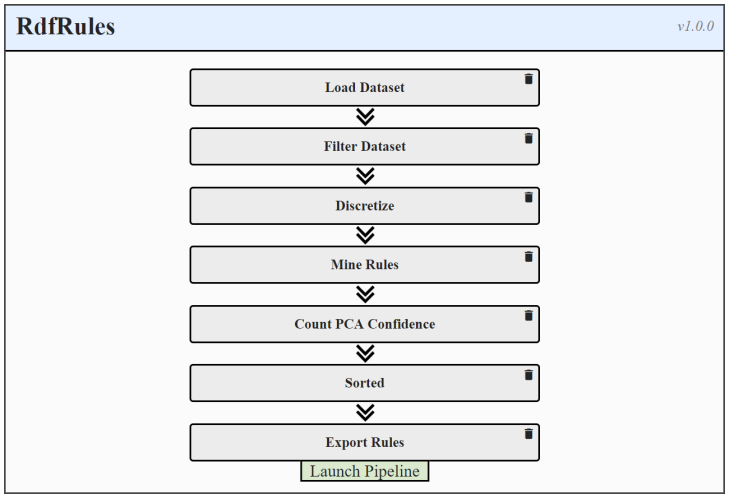


Fig. 4: An example of a mining workflow in the RdfRules GUI.

Scala API: a rule mining workflow

```scala
Dataset("yago.tsv")
  //we can filter triples by a condition.
  .filter(!_.triple.predicate.hasSameUriAs("participatedIn"))
  //we discretize all literals for "hasNumberOfPeople" predicate
  //into three equal-frequent bins.
  .discretize(DiscretizationTask.Equifrequency(3))
            (_.triple.predicate.hasSameUriAs("hasNumberOfPeople"))
  .mine(Amie()
  //enable to mine rules with constants.
  .addConstraint(RuleConstraint.WithInstances(true))
  //we define rules patterns - all rules must contain the "hasNumberOfPeople" predicate
  //at the head or body position.
  .addPattern(AtomPattern(predicate = Uri("hasNumberOfPeople")) =>: None)
  .addPattern(AtomPattern(predicate = Uri("hasNumberOfPeople"))))
  //rules post-processing: confidence counting and sorting.
  .computePcaConfidence(0.5)
  .sorted
  //we export mined rules into a file in a machine readable json format.
  .export("rules.json")
```

Java API: a rule mining workflow

```java
Dataset
  .fromFile("yago.tsv")
  .filter(quad -> !quad.getTriple().getPredicate().hasSameUriAs("participatedIn"))
  .discretize(new DiscretizationTask.Equifrequency(3),
              quad -> quad.getTriple().getPredicate().hasSameUriAs("hasNumberOfPeople"))
  .mine(RulesMining.amie()
  .withInstances(true)
  .addPattern(RulePattern.create().prependBodyAtom(
    new AtomPattern().withPredicate(new Uri("hasNumberOfPeople"))))
  .addPattern(RulePattern.create(
    new AtomPattern().withPredicate(new Uri("hasNumberOfPeople")))))
  .computePcaConfidence(0.5)
  .sorted()
  .export("rules.json")
```

Web Service: JSON task definition

```
[
  { "name": "LoadDataset", "parameters": { "path": "yago.tsv" } },
  { "name": "FilterQuads", "parameters": {
    "or": [{ "predicate": "<participatedIn>", "inverse": true }]
  }},
  { "name": "Discretize", "parameters": {
    "task": { "name": "EquifrequencyDiscretizationTask", "bins": 3 },
    "predicate": "<hasNumberOfPeople>"
  }},
  { "name": "Mine", "parameters": {
    "constraints": ["WithInstancesOnlyObjects"],
    "patterns": [{
      "head": { "predicate": { "name": "Constant", "value": "<hasNumberOfPeople>" } }
    },{
      "body": [{ "predicate": { "name": "Constant", "value": "<hasNumberOfPeople>" } }]
    }]
  }},
  { "name": "ComputePcaConfidence", "parameters": { "min": 0.5 } },
  { "name": "Sorted", "parameters": null },
  { "name": "ExportRules", "parameters": { "path": "rules.json" } }
]
```

Additional examples demonstrating the three available interfaces for RdfRules: Scala API, Java API, and the web service can be found in the RdfRules GitHub repository.[7]

5 Related Work

The RdfRules framework is partially built on the *EasyMiner.eu* data mining system [9]. That is a complex web application for association rule mining, outlier detection and rule-based classification. It offers a graphical user interface and a public REST API. The system is also able to mine rules from RDF-style datasets in two modes. The first one only transforms input RDF dataset into transactions of items and uses the common apriori-based algorithm for rules enumeration. The second solution uses AMIE+ approach only with basic measures without further extensions. This module does not provide any options for data pre-processing and post-processing.

Another system for RDF data processing is called SANSA-Stack [6]. This project offers a set of algorithms for distributed data processing of large-scale RDF knowledge bases. The implementation is adapted for Apache Spark environment. Beside classification and clustering algorithms it also contains methods for logical rules mining by the AMIE+ algorithm. SANSA-Stack is composed of several libraries and is considered as a framework appropriate for further use in others data mining systems. Hence, it does not contain any GUI or public endpoint.

Beside the state-of-the-art AMIE+ approach there are other algorithms and prototypes which come with new measures or methods. First, the SWARM algorithm, proposed by Barati et al. in [2], mines so-called semantic association rules. It uses the *rdf:type* predicate and the *rdfs:subClassOf* property to find

[7] https://github.com/propi/rdfrules/tree/master/experiments

rules with context to classes defined in an RDF schema or an ontology. Second, Tanon et al. in [7] introduced new scoring functions for better quality measurement of rules extracted from the RDF knowledge graphs with respect to the *Open World Assumption*. All these approaches can extend the current state of the RdfRules framework. Hence, the engine is adapted and open to add new measures or mining approaches in the future.

6 Conclusion and Future Work

RdfRules is a software system providing an end-to-end solution for rule mining over RDF knowledge graphs, implementing the state-of-the-art AMIE+ [5] algorithm. RdfRules covers the complete data mining lifecycle, most importantly providing functionality for data pre-processing, which are not supported by the first (and to our knowledge the only other) AMIE+ implementation made available by the AMIE+ authors.[8] What is unique to RdfRules is also a set of algorithmic extensions to AMIE+ that allow for faster mining and more concise results. The framework offers several interfaces to control a mining workflow and is suitable both for developers and for data analysts.

As to the work-in-progress and future work, we currently work on benchmarking RdfRules to evaluate the impact of the the new type of state space restrictions introduced in RdfRules on processing time. A promising direction for extending RdfRules is adding support for RDF schemas and ontologies, which would involve resource types with hierarchies into the mining process. Although the system currently supports multi-threading on a single machine, we would also like to add support for distributed mining and memory scaling on multiple nodes. Finally, AMIE+ produces logical rules with possibly complex structure, which may be found difficult to understand by some users. From the user perspective, research into human-perceived interpretability of logical rules is urgently needed.

7 Acknowledgements

This research was partly supported by grant IGA 33/2018 and institutional support for research activities of the Faculty of Informatics and Statistics, University of Economics, Prague.

References

1. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A Nucleus for a Web of Open Data. In: The Semantic Web. pp. 722–735. Springer, Berlin, Heidelberg (2007)

[8] https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/amie/

2. Barati, M., Bai, Q., Liu, Q.: Mining Semantic Association Rules from RDF Data. Knowledge-Based Systems **133**, 183–196 (2017)
3. Dougherty, J., Kohavi, R., Sahami, M.: Supervised and Unsupervised Discretization of Continuous Features. In: Prieditis, A., Russell, S. (eds.) Machine Learning Proceedings 1995, pp. 194–202. Morgan Kaufmann, San Francisco (CA) (1995)
4. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. pp. 226–231. KDD'96, AAAI Press (1996)
5. Galárraga, L., Teflioudi, C., Hose, K., Suchanek, F.M.: Fast Rule Mining in Ontological Knowledge Bases with AMIE+. The VLDB Journal **24**(6), 707–730 (2015)
6. Lehmann, J., Sejdiu, G., Bühmann, L., Westphal, P., Stadler, C., Ermilov, I., Bin, S., Chakraborty, N., Saleem, M., Ngomo, A.C.N., et al.: Distributed Semantic Analytics Using the SANSA Stack. In: International Semantic Web Conference. pp. 147–155. Springer (2017)
7. Pellissier Tanon, T., Stepanova, D., Razniewski, S., Mirza, P., Weikum, G.: Completeness-Aware Rule Learning from Knowledge Graphs. In: International Semantic Web Conference. pp. 507–525. Springer (10 2017)
8. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: A Core of Semantic Knowledge. In: Proceedings of the 16th International Conference on World Wide Web. pp. 697–706. WWW '07, ACM, New York, NY, USA (2007)
9. Vojíř, S., Zeman, V., Kuchař, J., Kliegr, T.: EasyMiner.eu: Web Framework for Interpretable Machine Learning Based on Rules and Frequent Itemsets. Knowledge-Based Systems **150**, 111–115 (2018)
10. Wirth, R., Hipp, J.: CRISP-DM: Towards a Standard Process Model for Data Mining. In: Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining. pp. 29–39. Citeseer (2000)
11. World Wide Web Consortium and others: RDF 1.1 Concepts and Abstract Syntax (2014)