RuDiK: Rule Discovery in Knowledge Bases

Stefano Ortona[§] Venkata Vamsikrishna Meduri[±] Paolo Papotti[∞]
[§] Meltwater [±] Arizona State University [∞] EURECOM
stefano.ortona@meltwater.com vmeduri@asu.edu papotti@eurecom.fr

ABSTRACT

RuDiK is a system for the discovery of declarative rules over knowledge-bases (KBs). RuDiK discovers both *positive* rules, which identify relationships between entities, e.g., "if two persons have the same parent, they are siblings", and *negative* rules, which identify data contradictions, e.g., "if two persons are married, one cannot be the child of the other". Rules help domain experts to curate data in large KBs. Positive rules suggest new facts to mitigate incompleteness and negative rules detect erroneous facts. Also, negative rules are useful to generate negative examples for learning algorithms. RuDiK goes beyond existing solutions since it discovers rules with a more *expressive rule language* w.r.t. previous approaches, which leads to wide coverage of the facts in the KB, and its mining is robust to existing *errors and incompleteness in the KB*. The system has been deployed for multiple KBs, including Yago, DBpedia, Freebase and WikiData, and identifies new facts and real errors with 85% to 97% accuracy, respectively. This demonstration shows how RuDiK can be used to interact with domain experts. Once the audience pick a KB and a predicate, they will add new facts, remove errors, and train a machine learning system with automatically generated examples.

PVLDB Reference Format:
Stefano Ortona, Vamsi Meduri, Paolo Papotti. RuDiK: Rule Discovery in Knowledge Bases. *PVLDB*, 11 (12): 1946 - 1949, 2018.
DOI: https://doi.org/10.14778/3229863.3236231

1. INTRODUCTION

Building large RDF knowledge-bases (KBs) is a popular trend in information extraction. Significant effort has been put on KBs creation in the last 10 years in the research community (e.g., Yago), as well as in the industry [1].

Unfortunately, due to their creation process, KBs are usually erroneous and incomplete. KBs are bootstrapped by extracting information from sources with minimal or no human intervention. Automation brings large scale, but also introduces noisy data to the KBs, as incorrect facts are propagated from the sources or introduced by the extractors [1]. Also, most KBs do not limit the information of interest with a schema that defines instance data, and let users add facts defined on new predicates by simply inserting new triples. Since *closed world assumption* (CWA) does not hold in KBs [1, 2], a missing fact is considered *unknown* rathen than false (*open world assumption*).

As a consequence, the amount of errors and incompleteness in KBs can be significant, with up to 30% errors for facts derived from the Web. Since KBs are large, e.g., WIKIDATA has more than 1B facts and 300M different entities, checking all triples to find errors or to add new facts cannot be done manually. A popular curation approach is to execute *rules* to improve data quality [2]. We consider two types of rules: (i) *positive rules* to enrich the KB with new facts and thus increase its coverage; (ii) *negative rules* to spot logical inconsistencies and identify erroneous triples.

Example 1: Consider a KB with information about parent and child relationships. A positive rule is the following:

$$r_1 : \texttt{parent}(b, a) \Rightarrow \texttt{child}(a, b)$$

stating that if a person a is parent of person b, then b is child of a. A negative rule has similar form, but different semantics. For example (*DOB* stands for Date Of Birth):

$$r_2 : \texttt{DOB}(a, v_0) \wedge \texttt{DOB}(b, v_i) \wedge v_0 > v_i \wedge \texttt{child}(a, b) \Rightarrow \bot$$

states that person b cannot be child of a if a was born after b. By querying the KB with r_2, we identify noisy triples stating that a child is born before one of her parents.

To be enforced over a KB, rules must be manually crafted, a task that can be difficult for domain experts without a CS background. Also, the rule creation process is usually expensive, as large KBs can have thousands of rules.

We demonstrate RuDiK (Rule Discovery in Knowledge Bases) [3], a rule discovery system[1]. Its features include:

1. Automatic discovery and selection of both *positive and negative rules* in a GUI.
2. *Numerical and string value comparisons* in the rules, which enable a large number of patterns to be expressed in the rules.
3. Realistic assumptions on the quality of the given KB: rules can be discovered over *noisy and incomplete KBs*.
4. *Incremental algorithms*, which greedily materialize the KB as a graph with low memory footprint, enabling execution on commodity machines.

[1] https://github.com/stefano-ortona/rudik

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236231

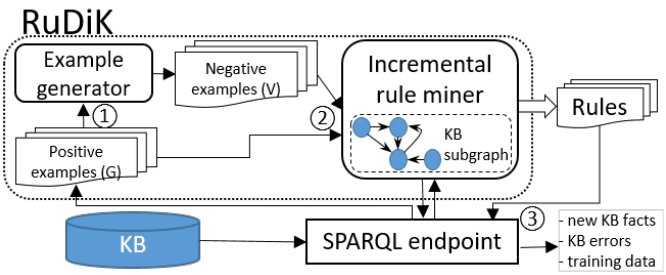

Figure 1: RuDiK architecture.

2. RuDiK SYSTEM OVERVIEW

Figure 1 shows the system architecture of RuDiK. We give an overview of the modules, and then describe the main methods (more details are in the full paper [3]). The system input is a KB and one of its predicates (e.g., `married`).

(1) Example Generation. Rules are mined from positive and negative examples for the given predicate (e.g., a set of pairs of person who are married and a set for persons who are not). Examples determine the quality of the rules, but crafting a large number of examples is a tedious and expensive exercise. The example generation module takes care of this task. It is initialized by positive examples from the KB and creates negative examples with an approach that is robust to missing data and errors in the KB.

(2) Incremental Rule Miner. The input of the module are positive and negative examples for the input predicate. In contrast to the traditional ranking of a large set of rules based on a measure of support [2], our module identifies a small set of *approximate* rules, i.e., rules that do not necessarily hold over all the examples, since data errors and incompleteness are in the nature of KBs. A greedy algorithm incrementally materializes the KB as a graph by navigating only the paths that potentially lead to promising rules, thus minimizing disk-access and memory footprint. These techniques enable the mining with a complex rule language.

(3) Rules Execution. Good rules in the output cover several input positive examples, and as few input negative examples as possible. Rules are then executed over the KB as SPARQL queries to produce new facts, identify inconsistencies, and generate training data for learning frameworks.

2.1 Generating Negative Examples

Given a KB kb and a target predicate $rel \in kb$ (e.g., `child`), we create two sets of examples. The first, namely the generation set G, consists of positive examples for rel, i.e., all pairs of entities (x, y) such that $\langle x, rel, y\rangle \in kb$. The second, namely the validation set V, contains counterexamples for rel. Differently from classic databases, we cannot assume that a fact that is not in a KB is false (closed world assumption), thus everything that is not stated is *unknown* (open world assumption). This implies that we must be careful in generating negative examples (truly false facts).

One simple way of creating false facts is to randomly select pairs from the Cartesian product of the entities. While this process gives negative examples with high precision, only a very small fraction of these entity pairs are *semantically related*. This issue has effects in the applications that consume the generated negative examples. In fact, unrelated entities may have no meaningful relationships. If there are no semantic patterns in the examples, this is reflected in lower quality in the generated rules. To generate negative examples that are correct and that are semantically related, we identify the entities that are more likely to be complete, i.e., entities for which the KB contains full information. This is done by exploiting the notion of *Local-Closed World Assumption* (*LCWA*) [1, 2]. LCWA states that if a KB contains one or more object values for a given subject and predicate, then it contains all possible values. Under this assumption, we identify entities that are likely to be complete and generate negative examples by taking the union of entities satisfying the LCWA. For example, if rel = `child`, a negative example is a pair (x, y) s.t. x has some children in the KB who are not y, or y is the child of someone who is not x. Moreover, for a candidate negative example over entities (x, y), we require that x must be connected to y via a predicate that is different from the target one. In other words, given a KB kb and a predicate rel, (x, y) is a negative example if $\langle x, rel', y\rangle \in kb$, with $rel' \neq rel$. This restriction guarantees that, for every $(x, y) \in V$, x and y are semantically related.

2.2 Rule Mining: Problem Formulation

Our goal is to automatically discover *Horn Rules* with universally quantified variables only. A Horn Rule is a disjunction of *atoms* with at most one unnegated atom. An atom is a predicate connecting two variables, two entities, an entity and a variable, or a variable and a constant.

We define the discovery problem for a single *target predicate* in the KB given as input (e.g., `child`). To obtain all rules for a KB, we compute rules for every predicate in it. We characterize a predicate with two sets of examples (i.e., pairs of entities). The generation set G contains examples for the target predicate, while the validation set V contains counterexamples for the same. Consider the discovery of positive rules for the `child` predicate; G contains true pairs of parents and children and V contains pairs of people who *are not* in a child relation. If we want to mine negative rules, the sets of examples are the same, but they switch role. To discover negative rules for `child`, G contains pairs of people not in a child relation and V contains true pairs.

An *exact* solution for our mining problem is composed by the minimal set of rules that covers all pairs in G and none of the pairs in V. We minimize the number of rules in the output to avoid overfitting rules covering only one pair, as such rules have no impact when applied to the KB.

Example 2: Consider the discovery of positive rules for predicate `couple` between two persons using as example the Obama family. G contains a positive example (Michelle, Barack), and V two negative examples with their daughters (Malia, Natasha). Given three rules:

$$r_3 : \texttt{livesIn}(a, v_0) \wedge \texttt{livesIn}(b, v_0) \Rightarrow \texttt{couple}(a, b)$$

$$r_4 : \texttt{hasChild}(a, v_i) \wedge \texttt{hasChild}(b, v_i) \Rightarrow \texttt{couple}(a, b)$$

$$r_5 : \texttt{hasChild}(Michelle, Malia) \wedge \texttt{hasChild}(Barack, Malia) \Rightarrow \texttt{couple}(Michelle, Barack)$$

Rule r_3 states that two persons are a couple if they live in the same place, while rule r_4 states that they are a couple if they have a child in common. Assuming the information `livesIn` and `hasChild` are in the KB, both rules r_3 and r_4 cover the positive example. Rule r_4 is an exact solution, as it does not cover any negative example, while this is not true for r_3, as also the daughters live in the same place. Rule r_5, which explicitly mentions entity values (constants), is also an exact solution, but it applies only for the positive example, i.e., it does not imply new facts from the KB.

If any of the `hasChild` relationships between the parents and the daughters is missing in G, the exact discovery would find only r_5 as a solution. This highlights that the exact discovery is not robust to data problems in KBs. Even if a valid rule exists semantically, missing triples or errors for the examples in G and V can lead to faulty coverage.

Given errors and missing information in KBs, we drop the requirement of exactly covering the sets with the rules. However, valid rules should cover examples in G, while covering elements in V can be an indication of incorrect rules. We model this idea as a *weight* associated with every rule. Weights enable the modeling of the presence of errors in KBs. Consider the case of negative rule discovery, where V contains positive examples from the KB. We will show in the demonstration several negative rules with significant coverage over V, which corresponds to errors in the KB. The approximate version of the discovery problem aims to identify rules that cover most of the elements in G and as few as possible elements in V. Since we do not want overfitting rules, we do not generate in R rules having constants only. We can map this problem to the *weighted set cover problem*.

2.3 Rule Mining: Incremental Algorithm

Our goal is to discover a set of rules to produce an optimal weighted set cover for the given examples. We define the *weight* for a rule to capture its quality w.r.t. G and V: the better the rule, the lower the weight. The weight is made of two components normalized by a parameter α. The first component captures the coverage over the generation set G – the ratio between the coverage of r over G and G itself. The second component quantifies the coverage of r over V. Parameter α defines the relevance of the components: a high value steers the discovery towards rules with high precision, while a low value champions the recall.

The weight definition is extended to define a *marginal weight* that quantifies the weight increase by adding a new rule r to a set of rules R. Since the weighted set cover problem aims at minimizing the total weight, we never add a rule to the solution if its marginal weight is greater than or equal to 0. The greedy solution guarantees a $\texttt{log}(k)$ approximation to the optimal solution, where k is the largest number of elements covered in G by a rule r in R.

The greedy algorithm for weighted set cover assumes that the universe of rules R has been generated (sets are available). One way to generate R is to translate the KB into a directed graph and map a valid rule to a path in the graph from a node x to a node y, for every pair $(x, y) \in G$. Clearly, computing all possible paths is prohibitive, as their number is very large. We avoid the generation of the universe R by considering at each iteration the most promising path on the graph. Inspired by the A^* algorithm, for each example $(x, y) \in G$, we start the navigation from x. We keep a queue of candidate paths, and at each iteration we expand the path with the minimum marginal weight (admissible heuristic for A* formulation). Whenever a path becomes valid, we add the corresponding rule to the solution and we do not expand it any further. The algorithm keeps looking for plausible paths until one of the termination conditions of the greedy cover algorithm is met.

The simultaneous rule *generation* and *selection* brings multiple benefits. First, we do not generate the entire graph for every example in G. Nodes and edges are generated *on demand*, whenever the algorithm requires their navigation.

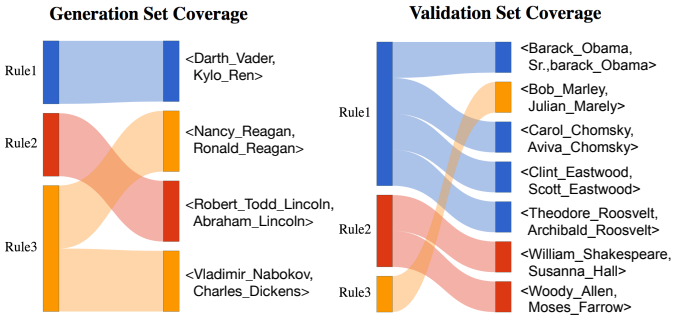

Figure 2: Coverage of three negative rules for predicate `child` over generation and validation sets.

Second, the weight estimation leads to pruning unpromising rules. If a rule does not cover new elements in G or in V, then it is pruned. These benefits enable us to enlarge the search space and consider also predicate atoms in the rules with literal comparisons beyond equalities. To discover such atoms, we materialize edges that connect literals with symbols from $\{<, \leqslant, \neq, >, \geqslant\}$, e.g., an edge '$<$' from a node "*March 31, 1930*" to a node "*March 21, 1986*". Unfortunately, the original KBs do not contain this information explicitly, and materializing such edges among all literals is infeasible. However, since our algorithm discovers paths for a pair of entities from G in isolation, the size of the graph resulting for one pair of entities is orders of magnitude smaller than the KB, thus we can compute all literal pairwise comparisons within a single example. Besides equality comparisons, we add '$>$','$\geqslant$','$<$','$\leqslant$' relationships between numbers and dates, and $\neq$ between all literals. This language allows us to discover rules such as the one stating that only Americans can become the president of U.S.A.:

$$\texttt{bornIn}(a, x) \wedge x \neq \textit{U.S.A.} \Rightarrow \neg\texttt{president}(a, \textit{U.S.A.})$$

3. DEMONSTRATION SCENARIOS

The audience will be able to use RuDiK to discover positive and negative rules from four popular and widely used KBs, namely DBPedia, Yago, Freebase, and Wikidata[2]. Participants will analyze the automatically discovered rules with a debugger that visualizes the coverage of the examples for each rule in the output. After selecting a rule, this can be executed to generate new facts or to identify contradictions in the KB. Users can assess the new facts and the errors, and compare the outcome of rules coming from different mining configurations. We now present the demonstration scenarios we will use to show the main features of RuDiK.

Rule Mining. First, the users will select a KB, a predicate of interest, such as `couple` or `founder`, and the kind of rules they want to discover (positive or negative). This triggers the generation of the negative examples and the automatic discovery of rules. The discovered rules will then be presented to the audience in their logic formalism in a result set. Every rule can be expanded to see more details, including the corresponding SPARQL query, an English translation, and statistics over its support over the given examples. In particular, the examples in the generation and validation sets are displayed with different coloring according to how they are covered by the rules in the output. Figure 2 shows

[2]available at `wiki.dbpedia.org`, `www.yago-knowledge.org`, `developers.google.com/freebase`, `www.wikidata.org`

the coverage of three negative rules for the `child` predicate in DBPEDIA (e.g., Rule 1 corresponds to r_2 in Section 1). By hovering the mouse over a rule, users can isolate and highlight examples covered just by the rule. This visualization enables a debugging of the generation process and highlights the pivoting role played by the examples. Every example can be expanded with a click to visualize the other entities involved in the rule(s) execution.

The interface will also be used to show the impact of different configuration settings, as discussed next.

Role of LCWA. We will show the impact of the LCWA assumption for the generation of negative examples. Given a predicate p, users will be able to test three generation strategies: RuDiK strategy (as detailed in Section 2.1), Random (randomly select k pairs (x, y) from the Cartesian product s.t. triple $\langle x, p, y\rangle \notin kb$), and LCWA (RuDiK strategy without the constraint that x and y must be connected). The results will show that *Random* and *LCWA* have both very high precision, but RuDiK examples lead to better rules. This is because randomly picked examples from the Cartesian product of subject and object often involve entities from very different time periods, and negative rules pivoting on time constraints are usually correct. Instead, by forcing x and y to be connected with a different predicate, RuDiK generates semantically related examples that lead to more rules. Negative rules such as $\texttt{parent}(a, b) \Rightarrow \texttt{notSpouse}(a, b)$ are not generated with random strategies, since the likelihood of randomly picking (in G) two people that are in a parent relation is very low. We will show that the RuDiK strategy enables the discovery of more types of rules, and not only rules involving time constraints.

Role of Set Cover. Our problem formulation leads to a concise set of rules in the output, which is preferable to the large set of rules obtained with a ranking based solution. Users will be able to verify this by obtaining the output rules according to their weight, without the set cover processing. Results will show that correct rules oftentimes are not ranked among the top-10, and that some meaningful rules are below the 100^{th} position. For example, the only valid negative rule for the predicate `founder`, which states that a person born after the company was founded cannot be its founder, figures at a rank of 127 when emitted by the ranking-based version of RuDiK, whereas it is included in the compact set discovered by the default version of RuDiK.

Role of Literals. One of the main features of RuDiK is the use of literals comparisons in the rules. To show that including literal values has a considerable impact on accuracy, both for positive and negative rules, we will allow users to enable and disable it. The effect will be particularly evident for negative rules, where rules without literals find less than half potential errors and with a lower precision. For predicate `founder`, we will show that a negative rule with literals discovers 79 potential errors on DBPEDIA with a 95% precision, while no errors are detected with rules without literals.

Role of Noise. Since our methods are designed to be robust, we will let attendees inject erroneous examples in the system input. While small amount of errors (up to 10%) have little effect on the quality of the results, increasing the noise percentage leads the algorithm to start discovering incorrect rules. However, we will show that even with more than 50% error rate, correct rules are discovered. Moreover, the precision of the results can be improved by tuning the parameter α to obtain rules that favour precision over recall.

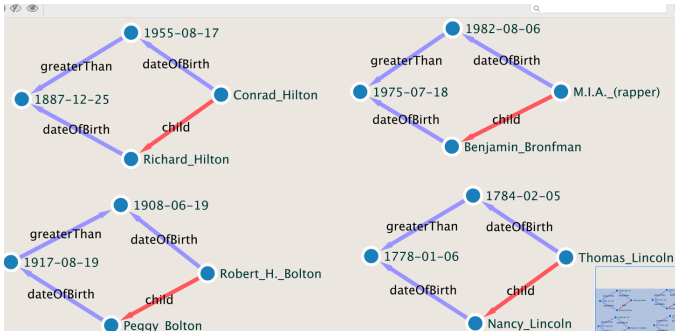

Figure 3: Rule Execution over the KB.

Rule Execution. Once a rule in the output is selected, users can run it on the KB to either infer new facts or to identify inconsistencies. As shown in Figure 3, for each output, the left-hand side instantiation of the rule will be displayed, effectively showing the provenance for each result. In the figure, we see the execution of the negative rule r_2 of Example 1, which covers 4 target examples in DBPEDIA. More specifically, the target predicate (`child`) is displayed with a red arrow, while alternative paths on the graph representing the left-hand side of the rule are shown with purple edges. This graphical representation enables users not only to quickly identify erroneous triples in the KB, but also to visually materialize paths on the graph that lead to the generation of such rules. By clicking on an entity, users can expand its surrounding graph. Moreover, users can select alternative paths and verify the coverage of the resulting rules against the input examples. For each rule, we report statistics such as the size of the sub-graph generated by the rule and the time spent for generation and validation queries.

Finally, we will demonstrate how discovered negative rules provide Machine Learning frameworks with training examples of quality comparable to examples manually crafted by humans. The quality of the negative examples will be quantified by training the same Machine Learning system with different training sets and by measuring the ultimate quality on the same test data. For this scenario, we will use DeepDive, a ML framework for information extraction [4]. DeepDive extracts entities and relations from text articles via distant supervision. The key idea behind distant supervision is to use an external source of information (e.g., a KB) to provide training examples for a supervised algorithm. As KBs provide positive examples only, we will show how the quality of the output of DeepDive increases when trained with negative examples obtained with our rules.

4. REFERENCES

[1] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. *PVLDB*, 7(10):881–892, 2014.

[2] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. Fast rule mining in ontological knowledge bases with AMIE+. *The VLDB Journal*, 24(6):707–730, 2015.

[3] S. Ortona, V. Meduri, and P. Papotti. Robust discovery of positive and negative rules in knowledge-bases. In *IEEE ICDE*, 2018.

[4] J. Shin, S. Wu, F. Wang, C. De Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using DeepDive. *PVLDB*, 8(11):1310–1321, 2015.