Robust Discovery of Positive and Negative Rules in Knowledge-Bases

Stefano Ortona* Vamsi Meduri† Paolo Papotti‡
* Meltwater – stefano.ortona@meltwater.com
† Arizona State University – vmeduri@asu.edu
‡ Eurecom – papotti@eurecom.fr

***Abstract*—We present RuDiK, a system for the discovery of declarative rules over knowledge-bases (KBs). RuDiK discovers rules that express *positive* relationships between entities, such as "if two persons have the same parent, they are siblings", and *negative rules*, i.e., patterns that identify contradictions in the data, such as "if two persons are married, one cannot be the child of the other". While the former class infers new facts in the KB, the latter class is crucial for other tasks, such as detecting erroneous triples in data cleaning, or the creation of negative examples to bootstrap learning algorithms. The system is designed to: *(i)* enlarge the *expressive power* of the rule language to obtain complex rules and wide coverage of the facts in the KB, *(ii)* discover *approximate* rules (soft constraints) to be robust to errors and incompleteness in the KB, *(iii)* use disk-based algorithms, effectively enabling rule mining in commodity machines. In contrast with traditional ranking of all rules based on a measure of support, we propose an approach to identify the subset of useful rules to be exposed to the user. We model the mining process as an incremental graph exploration problem and prove that our search strategy has guarantees on the optimality of the results. We have conducted extensive experiments using real-world KBs to show that RuDiK outperforms previous proposals in terms of efficiency and that it discovers more effective rules for the application at hand.**

I. Introduction

Building large RDF knowledge-bases (KBs) is a popular trend in information extraction. KBs store information in the form of triples, where a *predicate*, expresses a binary relation between a *subject* and an *object*. KB triples, called facts, store information about real-world entities and their relationships, such as "Michelle Obama is married to Barack Obama". Significant effort has been put on KBs creation in the last 10 years in the research community (DBPedia [3], FreeBase [4], Wikidata [24], DeepDive [19], Yago [20]) as well as in the industry (e.g., Google [10], Wal-Mart [9]).

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. This leads to two main problems. First, false facts are propagated from the sources to the KBs, or introduced by the extractors [10]. Second, usually KBs do not limit the information of interest with a schema and let users add facts defined on new predicates by simply inserting new triples. Since *closed world assumption* (CWA) does no longer hold in KBs [10], [13], we cannot assume that a missing fact is false, but we rather label it as *unknown* (*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 [1], [21]. Since KBs are large, e.g., Wikidata has more than 1B facts and 300M entities, checking all triples to find errors or to add new facts cannot be done manually. A natural approach to assist curators is to discover *declarative rules* that can be executed over the KB to improve the quality of the data [2], [5], [13]. We target the discovery of 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 executing the rule as a query over `child` facts, we identify erroneous triples.

In order to be executed over a KB, or plugged into an existing inference system [17], 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 very expensive, as large KB can have rules in the thousands [22]. A rule discovery system is therefore a crucial asset to help the users in data curation. However, three main challenges arise when discovering positive and negative rules from KBs.

Data Quality. While traditional rule mining techniques assume that data is either clean or has a negligible amount of errors [6], KBs can present errors and are incomplete.

Open World Assumption. Other approaches rely on the presence of positive and negative examples [8], [16], but KBs contain only positive statements, and, without CWA, there is no immediate solution to derive counter examples.

Volume. Existing approaches for rule discovery assume that data fit into main memory [2], [13], [5], [12]. Given the large and increasing size of KBs, these approaches focus on a simple rule language to minimize the size of the search space.

We present RuDiK (Rule Discovery in Knowledge Bases), a novel system for the discovery of rules over KBs that addresses these challenges. RuDiK is the first system designed to discover both *positive and negative rules* over noisy and incomplete KBs. By relying on disk based algorithms, RuDiK can handle a larger search space and discover rules with a richer language that allows value comparisons. This increase in the *expressive power* enables a larger number of patterns to be expressed in the rules, and therefore a larger number of new facts and errors can be identified with high accuracy. These results are achieved by exploiting the following contributions.

1. Problem Definition. We formally define the problem of robust rule discovery over erroneous and incomplete KBs. The input of the problem are two sets of positive and negative examples for every predicate. In contrast to the traditional ranking of a large set of rules based on a measure of support [8], [13], [18], our problem definition aims at the identification of a subset 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. The solution is then the smallest set of rules that cover the majority of input positive examples, and as few input negative examples as possible (Section III).

2. Example Generation. Positive and negative examples for a target predicate are crucial to our approach as they determine the ultimate quality of the rules. However, crafting a large number of negative examples is a tedious exercise that requires manual work. We present an algorithm for example generation that is aware of missing data and inconsistencies in the KB. Our generated examples lead to better rules than examples obtained with alternative approaches (Section IV).

3. Rule Discovery Algorithm. We give a $\texttt{log}(k)$-approximation algorithm for the rule discovery problem, where k is the maximum number of input positive examples covered by a single rule. We discover rules by judiciously using the memory. The algorithm incrementally materializes the KB as a graph, and discovers rules by navigating only the paths that potentially lead to the best rules. By materializing only the portion of the KB that is needed for the promising rules, the disk-access is minimized and the low memory footprint enables the mining with a richer rule language (Section V).

We experimentally test the performance of RuDiK on three popular and widely used KBs. We show that our system delivers accurate rules, with a relative increase in average precision by 45% both in the positive and in the negative settings w.r.t. state-of-the-art systems. Also, differently from other proposals, RuDiK performs consistently well with KBs of all sizes on a regular laptop. Finally, we demonstrate how discovered negative rules provide Machine Learning algorithms with training examples of quality comparable to examples manually crafted by humans (Section VI).

II. Preliminaries

We focus on discovering rules from RDF KBs. An RDF KB is a database that represents information through RDF triples $\langle s, p, o\rangle$, where a *subject* (s) is connected to an *object* (o) via a *predicate* (p). Triples are often called *facts*. For example, the fact that Scott Eastwood is the child of Clint Eastwood could be represented through the triple $\langle Clint\_Eastwood, child, Scott\_Eastwood\rangle$. RDF KB triples respect three constraints: (i) triple subjects are always *entities*, i.e., concepts from the real world; (ii) triple objects can be either entities or *literals*, i.e., primitive types such as numbers, dates, and strings; (iii) triple predicates specify real-world relationships between subjects and objects.

Differently from relational databases, KBs usually do not have a schema that defines allowed instances, and new predicates can be added by inserting triples. This model allows great flexibility, but the likelihood of introducing errors is higher than traditional schema-guided databases. While KBs can include *T-Box* facts to define classes, domain/co-domain types for predicates, and relationships among classes to check integrity, in most KBs – including the ones used in our experiments – such information is missing. Hence our focus is on the *A-Box* facts that describe instance data.

A. Language

Our goal is to automatically discover first-order logical formulas in KBs. More specifically, we target the discovery of *Horn Rules* with universally quantified variables only. A Horn Rule is a disjunction of *atoms* with at most one unnegated atom. In the implication form, they have the following format:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow B$$

where $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ is the *body* of the rule (a conjunction of atoms) and B is the *head* of the rule (a single atom). However, it is logically equivalent to rewrite the atom in the head of the rule in its negated form in the body to emphasize contradictions:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge \neg B \Rightarrow \perp$$

We therefore distinguish between *positive rules*, which generate new facts (e.g., r_1 in Example 1), and *negative rules* (e.g., r_2 in Example 1), which identify incorrect facts, similarly to denial constraints for relational data [6]. An atom is a predicate connecting two variables, two entities, or an entity and a variable. For simplicity, we write an atom with the notation $\texttt{rel}(a, b)$, where $\texttt{rel}$ is a KB predicate and a, b are either variables or entities. Given a rule r, we define r_{body} and r_{head} as the body and the head of the rule, respectively, and refer to the variables in the head of the rule as the *target variables*.

We remark that we also discover rules with a body atom in its negated form in the head. The result is a formula that generates negative facts. For example, negative rule r_2 is obtained by rewriting in the body the atom $\texttt{notChild}$ in the following rule:

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

As shown in the negative rule, we allow *literal comparisons* in our rules. A literal comparison is a special atom $\texttt{rel}(a, b)$, where $\texttt{rel} \in \{<, \leqslant, \neq, >, \geqslant\}$, and a and b can only be assigned to literal values except if $\texttt{rel}$ is equal to $\neq$, i.e., we allow inequality comparisons for entities.

Given a KB kb and an atom $A = \texttt{rel}(a, b)$ where a and b are two entities, we say that A *holds* over kb iff $\langle a, \texttt{rel}, b\rangle \in kb$. Given an atom $A = \texttt{rel}(a, b)$ with at least one variable, we say that A can be *instantiated* over kb if there exists at least one entity from kb for each variable in A s.t. if we substitute all variables in A with these entities, A holds over kb. Transitively, we say that r_{body} can be instantiated over kb if every atom (with entities) in r_{body} can be instantiated and every literal comparison is logically true.

As in other approaches [13], [5], we want to avoid Cartesian products in our rules and therefore define a rule *valid* iff every variable in it appears at least twice. Target variables already appear once in the head of the rule, but each non target variable must be involved in a join or in a comparison.

B. Rule Coverage

Given a pair of entities (x, y) from a KB kb and a Horn Rule r, we say that r_{body} *covers* (x, y) if $(x, y) \models r_{body}$. In other words, given a rule $r : r_{body} \Rightarrow \texttt{r}(a, b)$, r_{body} covers a pair of entities $(x, y) \in kb$ iff we can substitute a with x, b with y, and the rest of the body can be instantiated over kb. Given a set of pair of entities $E = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ and a rule r, we denote by $C_r(E)$ the *coverage* of r_{body} over E as the set of elements in E covered by r_{body}: $C_r(E) = \{(x, y) \in E | (x, y) \models r_{body}\}$.

Given the body r_{body} of a rule r, we denote by r^*_{body} the *unbounded body* of r. The unbounded body of a rule is obtained by keeping only atoms that contain a target variable and substituting such atoms with new atoms where the target variable is paired with a new unique variable. As an example, given $r_{body} = \texttt{rel}_1(a, v_0) \wedge \texttt{rel}_2(v_0, b)$ where a and b are the target variables, $r^*_{body} = \texttt{rel}_1(a, v_i) \wedge \texttt{rel}_2(v_{ii}, b)$. While in r_{body} the target variables are bounded to be connected by variable v_0, in r^*_{body} they are unbounded. Given a set of pair of entities $E = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ and a rule r, we denote by $U_r(E)$ the *unbounded coverage* of r^*_{body} over E as the set of elements in E covered by r^*_{body}: $U_r(E) = \{(x, y) \in E | (x, y) \models r^*_{body}\}$. Note that, given a set E, $C_r(E) \subseteq U_r(E)$.

Example 2: We denote with E the set of all possible pairs of entities in kb. The coverage of r_2 of Example 1 over E ($C_r(E)$) is the set of all pairs of entities $(x, y) \in kb$ s.t. both x and y have the DOB information and x is born after y. The unbounded coverage of r over E ($U_r(E)$) is the set of all pairs of entities (x, y) s.t. both x and y have the DOB information, no matter what the relation between the two birth dates is.

The unbounded coverage is essential to distinguish between missing and inconsistent information: if for a pair of entities (x, y) the DOB is missing for either x or y, we cannot say whether x was born before or after y. But if both x and y have the DOB and x is born before y, we can state that r_2 does not cover (x, y). As KBs are incomplete, we must discriminate between missing and conflicting information. We extend the definition of coverage and unbounded coverage to a set of rules $R = \{r_1, r_2, \cdots, r_n\}$ as the union of individual coverages:

$$C_R(E) = \bigcup_{r \in R} C_r(E) \qquad U_R(E) = \bigcup_{r \in R} U_r(E)$$

III. Rule Discovery for Noisy KBs

For the sake of simplicity, we define the discovery problem for a single *target predicate* given as input. To obtain all rules for a given KB, we compute rules for every predicate in it. We characterize a predicate with two sets of pairs of entities. The *generation set* G contains examples for the target predicate, while the *validation set* V contains counter examples 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 identify errors (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 pairs of entities respecting the child relation.

We formalize next the *exact discovery problem*. In the following definitions, we assume for the sake of simplicity that all possible valid rules and the sets of examples have been already generated, we detail in the rest of the paper how they are efficiently obtained from the KB.

Definition 1: Given a KB kb, two sets of pairs of entities G and V from kb with $G \cap V = \emptyset$, and all the valid Horn Rules R for kb, a solution for the *exact discovery problem* is a subset R' of R s.t.:

$$\underset{R'}{\operatorname{argmin}}(size(R') | (C_{R'}(G) = G) \wedge (C_{R'}(V) \cap V = \emptyset))$$

The exact solution is the minimal set of rules that covers all pairs in G and none of the pairs in V. It minimizes the number of rules in the output ($size(R')$) to avoid overfitting rules covering only one pair, as such rules have no impact when applied on the KB. In fact, given a pair of entities (x, y), there is always an overfitting rule whose body covers only (x, y) by assigning target variables to x and y as shown next.

Example 3: Consider the discovery of positive rules for the predicate couple between two persons using as example the Obama family. A positive example is (Michelle, Barack) and a negative example is 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(x,y) and hasChild(x,y) 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 the negative example, while this is not true for r_3, as also the daughters live in the same place. Rule r_5 explicitly mentions entity values (constants) in its head and body. It is also an exact solution, but it applies only for the given positive example.

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. In the worst case, every rule in the exact solution would cover only one example in G, i.e., a set of overfitting rules with no effect when applied on the KB.

A. Weight Function

Given errors and missing information in both G and V, we drop the requirement of exactly covering the sets with the rules. In other words, we mine rules that hold for most of the data (soft-constraints), as we want to be robust w.r.t. noise and incompleteness. However, coverage is a strong indicator of quality: good rules should cover several examples in G, while covering elements in V can be an indication of incorrect rules. We model this idea in a *weight* associated with every rule.

Definition 2: Given a KB kb, two sets of pair of entities G and V from kb with $G \cap V = \varnothing$, and a Horn Rule r, the *weight of* r is defined as follow:

$$w(r) = \alpha \cdot (1 - \frac{\mid C_r(G) \mid}{\mid G \mid}) + \beta \cdot (\frac{\mid C_r(V) \mid}{\mid U_r(V) \mid}) \tag{1}$$

with $\alpha, \beta \in [0,1]$ and $\alpha + \beta = 1$, thus $w(r) \in [0,1]$.

The weight captures the quality of a rule w.r.t. G and V: the better the rule, the lower the weight – a perfect rule covering all generation elements of G and none of the validation elements in V has a weight of 0. The weight is made of two components normalized by parameters α and β. 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. The coverage over V is divided by the unbounded coverage of r over V, instead of the total elements in V, because some elements in V might not have the predicates stated in r_{body}. Intuitively, we restrict V with unbounded coverage to validate on "qualifying" examples.

Parameters α and β give relevance to each component. A high β steers the discovery towards rules with high precision by penalizing the ones that cover negative examples, while a high α champions the recall by favoring rules covering more generation examples.

Example 4: Consider again rule r_2 of Example 1 and two sets of pairs of entities G and V from a KB kb. The first component of w_r is computed as 1 minus the number of pairs (x, y) in G where x is born after y divided by the number of elements in G. The second component is the ratio between number of pairs (x, y) in V where x is born after y and number of pairs (x, y) in V where the birth date for both x and y is known in kb, i.e., examples with missing birth dates are not in $U_{r_2}(V)$.

Definition 3: Given a set of rules R, the *weight for* R is:

$$w(R) = \alpha \cdot (1 - \frac{\mid C_R(G) \mid}{\mid G \mid}) + \beta \cdot (\frac{\mid C_R(V) \mid}{\mid U_R(V) \mid})$$

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 report in the experimental evaluation several negative rules with significant coverage over V, which corresponds to errors in the KB. The weight is important also for plugging rules into existing inference systems for KBs. For example, weighted rules can be interpreted as soft constraints for probabilistic reasoning [17].

B. Problem Definition

We can now state the approximate version of the problem.

Definition 4: Given a KB kb, two sets of pair of entities G and V from kb with $G \cap V = \varnothing$, all the valid Horn Rules R for kb, and a w weight function for R, a solution for the *robust discovery problem* is a subset R' of R such that:

$$\underset{R'}{\operatorname{argmin}}(w(R') | C_{R'}(G) = G)$$

The *robust* version of the discovery problem aims to identify rules that cover all 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 in both target variables, thus avoiding any rule that covers only one example.

We can map this problem to the *weighted set cover problem*, which is proven to be NP–complete [7]. The reduction follows immediately from the following mapping: the set of elements (universe) corresponds to the generation examples in G, the input sets are identified by the rules defined in R (where each rule covers a subset of G), the non-negative weight function $w : r \rightarrow \mathbb{R}$ is $w(r)$ in Definition 2, and the cost of R is defined to be its total weight, according to Definition 3.

IV. Rule and Example Generation

In this Section we describe how to generate the universe of all possible rules. We start by assuming that the positive and the negative examples are given, and then show how they can be computed. However, our approach is independent of how G and V are generated: they could be manually crafted by domain experts, with significant additional manual effort.

We detail the discovery of positive rules having true facts in G and false facts in V. In the dual problem of negative rule discovery, our approach remains unchanged, we just switch the roles of G and V. The generation set G is formed out of false facts, while the validation set V is built from true facts.

A. Rule Generation

In the universe of all possible rules R, each rule must cover one or more examples from the generation set G. Thus the universe of all possible rules is generated by inspecting the elements of G alone. We translate a KB kb into a directed graph: entities and literals are the nodes, and there is a directed edge from node a to node b for each triple $\langle a, rel, b \rangle \in kb$. Edges are labelled with the relation rel that connects subject to object. Figure 1 shows four triples.

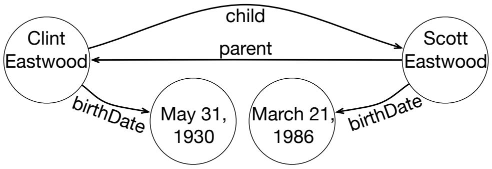

Fig. 1. Graph example for four triples from DBpedia.

The body of a rule can be seen as a path in the graph. In Figure 1, the body `child`(a, b) $\wedge$`parent`(b, a) corresponds to the path *Clint Eastwood* $\rightarrow$ *Scott Eastwood* $\rightarrow$ *Clint Eastwood*. As defined in Section II-A, a valid body contains target variables a and b at least once, every other variable at least twice, and atoms are transitively connected. If we allow navigation of edges independently of the edge direction, we can translate bodies of valid rules to valid paths on the graph. Given a pair of entities (x, y), a *valid body* corresponds to a valid path p on the graph such that: (i) p starts at the node x; (ii) p covers y at least once; (iii) p ends in x, in y, or in a different node that has been already visited. Given the body of a rule r_{body}, r_{body} covers a pair of entities (x, y) iff there exists a valid path on the graph that corresponds to r_{body}. This implies that for a pair of entities (x, y), we can generate bodies of all possible valid rules by computing all valid paths starting at x with a standard BFS. The key point is the ability to navigate each edge in any direction by turning the original directed graph into an undirected one. However, we need to keep track of the original direction of the edges. This is essential when translating paths to rule bodies. In fact, an edge directed from a to b produces the atom $\texttt{rel}(a, b)$, while b to a produces $\texttt{rel}(b, a)$.

Since every node can be traversed multiple times, for two entities x and y there might exist infinite valid paths starting from x. This is avoided with a $maxPathLen$ parameter that determines the maximum number of edges in the path, i.e., the maximum number of atoms allowed in the corresponding body of the rule. We show the impact of this parameter in Section VI.

We now describe the two main steps in our generation of the universe of all possible rules for G.

1. Create Paths. Given a pair of entities (x, y), we retrieve from the KB all nodes at a distance smaller than $maxPathLen$ from x or y, along with their edges. The retrieval is done recursively: we maintain a queue of entities, and for each entity in the queue we execute a SPARQL query against the KB to get all entities (and edges) at distance 1 from the current entity – we call these queries *single hop queries*. At the n-th step, we add the new found entities to the queue iff they are at a distance less than $(maxPathLen - n)$ from x or y and they have not been visited before. The queue is initialized with x and y. Given the graph for every (x, y), we then compute all valid paths starting from every x.

2. Evaluate Paths. Computing paths for every example in G implies also computing the coverage over G for each rule. The *coverage* of a rule r is the number of elements in G for which there exists a path corresponding to r_{body}. Once the universe of all possible rules has been generated (along with coverages over G), computing coverage and unbounded coverage over V requires only the execution of two SPARQL queries against the KB for each rule in the universe.

Since one of our goals is to increase the expressive power of discovered rules, we generate different atom types:

Literal comparison. We want predicate atoms with comparisons beyond equalities. To discover such atoms, the graph representation must contain edges that connect literals with one (or more) symbol from $\{<, \leqslant, \neq, >, \geqslant\}$. As an example, Figure 1 would contain an edge '$<$' from node "*March 31, 1930*" to node "*March 21, 1986*". Unfortunately, the original KB does not contain this kind of information explicitly, and materializing such edges among all literals is infeasible.

However, in our algorithm we discover paths for a pair of entities from G in isolation. The size of the graph resulting for a pair of entities is orders of magnitude smaller than the KB, thus we can afford to compare all literal pairwise comparisons within a single example graph. Besides equality comparisons, we add '$>$','$\geqslant$','$<$','$\leqslant$' relationships between numbers and dates, and $\neq$ between all literals. These new relationships are treated as normal atoms (edges): $x \geqslant y$ is equivalent to $\texttt{rel}(x, y)$, where $\texttt{rel}$ is equal to $\geqslant$.

Not equal variables. The "not equal" operator introduced for literals is useful for entities as well. Consider the rule:

$$\texttt{bornIn}(a, x) \wedge x \neq b \wedge \texttt{president}(a, b) \Rightarrow \perp$$

It states that if a person a is born in a country that is different from b, then a cannot be the president of b. One way to consider inequalities among entities is to add edges among all pairs of entities in the graph. However, this strategy is inefficient and would lead to many meaningless rules. To limit the search space while aiming at meaningful rules, we use the `rdf:type` triples associated to entities. We add an inequality edge in the input example graph only between pairs of entities of the same type (as in the example above).

Constants. Finally, we allow the discovery of rules with constant selections. Suppose that for the above president rule, all examples in G are people born in "*U.S.A.*", and there is at least one country for which this rule is not valid. According to our problem statement, the right rule is therefore:

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

To discover such atoms, we promote a variable v in a given rule r to an entity e iff for every $(x, y) \in G$ covered by r, v can always be instantiated with the same value e.

B. Input Example Generation

Given a KB kb and a predicate $rel \in kb$, we automatically build a generation set G and a validation set V as follows. G consists of positive examples for the target predicate rel, i.e., all pairs of entities (x, y) such that $\langle x, rel, y \rangle \in kb$. V consists of counter (negative) examples for the target predicate. These are more complicated to generate because of the open world assumption in KBs. Differently from classic databases, we cannot assume that what is not stated in a KB is false (closed world assumption), thus everything that is not stated is *unknown*. However, since the likelihood of two randomly selected entities being a positive example is extremely low, one simple way of creating false facts is to randomly select pairs from the Cartesian product of the entities [16]. While this process gives negative examples with a very high precision, only a very small fraction of these entity pairs are *semantically related*. This semantic aspect has effects in the applications that use the generated negative examples. In fact, unrelated entities have less meaningful paths than semantically related entities and this is reflected in lower quality in the experimental results.

A semantic connection is guaranteed for positive examples by definition, since pairs in G are always connected at least by the target predicate. To generate negative examples that are likely to be correct (true false facts) and that are semantically related, we mine the facts to identify the entities that are more likely to be complete, i.e., entities for which the KB contains full information. This process is done exploiting and extending the popular notion of *Local-Closed World Assumption* (*LCWA*) [13]. LCWA states that if a KB contains one or more object values for a given subject and predicate, then it contains all possible values. For example, if a KB contains one or more children of Clint Eastwood, then it contains all his children. This is always true for *functional* predicates (e.g., `capital`), while it might not hold for non-functional ones (e.g., `child`).

We generate negative examples taking the union of entities satisfying the LCWA. For a predicate rel, a negative example is a pair (x, y) where either x is the subject of one or more triples $\langle x, rel, y' \rangle$ with $y \neq y'$, or y is the object of one or more triples $\langle x', rel, y \rangle$ with $x \neq x'$. 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. The LCWA guarantees that, since at least another child exists for x, (x, y) cannot be in such relation and we can safely use the pair as a counter-example. In addition, to obtain examples that are semantically related, it is enough to add the constraint that every example is made from a pair of entities that are connected via a predicate different from the target predicate. In other words, given a KB kb and a target predicate rel, (x, y) is a negative example if $\langle x, rel', y \rangle \in kb$, with $rel' \neq rel$.

Example 5: A negative example (x, y) for the target predicate `child` has the following characteristics: *(i)* x and y are not connected by a `child` predicate; *(ii)* either x has one or more children (different from y) or y has one or more parents (different from x); *(iii)* x and y are connected by a predicate that is different from `child` (e.g., `colleague`).

To enhance the quality of the input examples and avoid cases of mixed types, we require that for every example pair (x, y), either in G or V, all the x occurrences have the same *type*, same for the y values.

V. Discovery Algorithm

We introduce a greedy approach to solve the approximate discovery problem (Section III-B). Since the number of possible rules can be very large, we introduce an algorithm that generates only promising rules from the KB, while preserving the same quality guaranteed by the exhaustive generation.

A. Marginal Weight for a Greedy Algorithm

Our goal is to discover a set of rules to produce a weighted set cover for the given examples. We therefore follow the intuition behind the greedy algorithm for weighted set cover by defining a *marginal weight* for rules that are not yet included in the solution [7].

Definition 5: Given a set of rules R and a rule r such that $r \notin R$, the marginal weight of r w.r.t. R is defined as:

$$w_m(r) = w(R \cup \{r\}) - w(R)$$

The marginal weight quantifies the weight increase by adding r to an existing set of rules. Since the 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.

If all rules have been generated, the algorithm for greedy rule selection is quite straightforward: given a generation set G, a validation set V, and the universe of all possible rules R, pick at each iteration the rule r with minimum marginal weight and add it to the solution R'. The algorithm stops when one of the following termination conditions is met: *1)* R is empty – all the rules have been included in the solution; *2)* R' covers all elements of G; *3)* the minimum marginal weight is greater than or equal to 0, i.e., among the remaining rules in R, none of them has a negative marginal weight.

The greedy solution guarantees a `log(k)` approximation to the optimal solution [7], where k is the largest number of elements covered in G by a rule in R. If the optimal solution is made of rules that cover disjoint sets over G, then the greedy solution coincides with the optimal one.

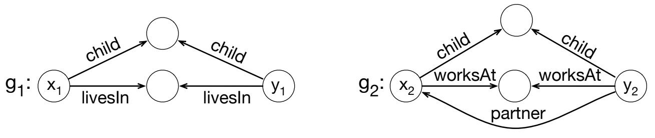

Fig. 2. Two positive examples.

B. A^* Graph Traversal

The greedy algorithm for weighted set cover assumes that the universe of rules R has been generated. To generate R, we need to traverse all valid paths from a node x to a node y, for every pair $(x, y) \in G$. But do we need all possible paths for every example?

Example 6: Consider the mining of positive rules for the target predicate `spouse`. The generation set G includes two examples g_1 and g_2 shown as graphs in Figure 2. Assume for simplicity that all rules in the universe have the same coverage and unbounded coverage over the validation set V. One candidate rule is r : $\texttt{child}(x, v_0) \wedge \texttt{child}(y, v_0) \Rightarrow \texttt{spouse}(x, y)$, stating that entities x and y with a common child are married. In the graph, r covers both g_1 and g_2. Since all rules have the same coverage and unbounded coverage over V, there is no need to generate any other rule. In fact, any other candidate rule will not cover new elements in G, therefore their marginal weights will be greater than or equal to 0. Thus the creation and navigation of edges `livesIn` in g_1, `worksAt` in g_2, and `partner` in g_2 is not needed.

Based on the above observation, we avoid the generation of the entire universe R, but rather consider at each iteration the most promising path on the graph as in the A^* graph traversal algorithm [14]. For each example $(x, y) \in G$, we start the navigation from x. We keep a queue of not valid rules, and at each iteration we consider the rule with the minimum marginal weight, which corresponds to paths in the example graphs. We expand the rule by following the edges, and we add the new founded rules to the queue of not valid rules. Unlike A^*, we do not stop when a rule (path) reaches the node y (i.e., becomes valid). Whenever a rule becomes valid, we add the 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.

A crucial point in A^* is the definition of the estimation cost. To guarantee the solution to be optimal, the estimation must be *admissible* [14], i.e., the estimated cost must be less than or equal to the actual cost. In our setting, given a rule that is not yet valid and needs to be expanded, we define an admissible estimation of the marginal weight.

Definition 6: Given a rule $r : A_1 \wedge A_2 \cdots A_n \Rightarrow B$, we say that a rule r' is an *expansion* of r iff r' has the form $A_1 \wedge A_2 \cdots A_n \wedge A_{n+1} \Rightarrow B$.

In the graph traversal, expanding r means traversing one further edge on the path defined by r_{body}. To guarantee the optimality condition, the estimated marginal weight for a rule r that is not valid must be less than or equal to the actual

Algorithm 1: RUDIK Rule Discovery.

```
   input : $G$ – generation set
   input : $V$ – validation set
   input : $maxPathLen$ – maximum rule body length
   output: $R_{opt}$ – union of rules in the solution
1  $R_{opt} \leftarrow \varnothing$;
2  $N_f \leftarrow \{x|(x,y) \in G\}$;
3  $Q_r \leftarrow$ expandFrontiers($N_f$);
4  $r \leftarrow \underset{r \in Q_r}{\text{argmin}}(w^*_m(r))$;
5  repeat
6      $Q_r \leftarrow Q_r \backslash \{r\}$;
7      if isValid($r$) then
8          $R_{opt} \leftarrow R_{opt} \cup \{r\}$;
9      else
           // rules expansion
10         if length($r_{body}$) $< maxPathLen$ then
11             $N_f \leftarrow$ frontiers($r$);
12             $Q_r \leftarrow Q_r \cup$ expandFrontiers($N_f$);
13     $r \leftarrow \underset{r \in Q_r}{\text{argmin}}(w^*_m(r))$;
14 until $Q_r = \varnothing \vee C_{R_{opt}}(G) = G \vee w^*_m(r) \geqslant 0$;
15 return $R_{opt}$
```

weight of any valid rule that is generated by expanding r. Given a rule and some expansions of it, we can derive the following.

Lemma 1: *Given a rule r and a set of pair of entities E, then for each r' expansion of r, $C_{r'}(E) \subseteq C_r(E)$ and $U_{r'}(E) \subseteq U_r(E)$.*

The above Lemma states that the coverage and unbounded coverage of an expansion r' of r are contained in the coverage and unbounded coverage of r, respectively, and directly derives from the augmentation inference rule for functional dependencies. The only positive contribution to marginal weights is given by $|C_{R \cup \{r\}}(V)|$. $|C_{R \cup \{r\}}(V)|$ is equivalent to $|C_R(V)| + |C_r(V) \backslash C_R(V)|$, thus if we set $|C_r(V) \backslash C_R(V)| = 0$ for any r that is not valid, we guarantee an admissible estimation of the marginal weight. We estimate the coverage over the validation set to be 0 for any rule that can be further expanded, since expanding it may bring the coverage to 0.

Definition 7: Given a *not valid* rule r and a set of rules R, we define the *estimated marginal weight* of r as:

$$w^*_m(r) = -\alpha \cdot \frac{|C_r(G) \backslash C_R(G)|}{|G|} + \beta \cdot \left(\frac{|C_R(V)|}{|U_{R \cup \{r\}}(V)|} - \frac{|C_R(V)|}{|U_R(V)|}\right)$$

The estimated marginal weight for a valid rule is equal to the actual marginal weight (Definition 5). Valid rules are not considered for expansion, therefore we do not need to estimate their weights since we know the actual ones. Given Lemma 1, we can see that $w^*_m(r) \leqslant w^*_m(r')$, for any r' expansion of r. Thus our marginal weight estimation is admissible.

We are ready to introduce Algorithm 1, which shows the modified set cover procedure, including the A^*-like rule generation. For a rule r, we call *frontier nodes*, $N_f(r)$, the last visited nodes in the paths that correspond to r_{body} from every example graph covered by r. Expanding a rule r implies navigating a single edge from any frontier node. In the algorithm, the set of frontier nodes is initialized with starting nodes x, for every $(x,y) \in G$ (Line 2). The algorithm maintains a queue of rules Q_r, from which it chooses at each iteration the rule with minimum estimated weight. The function `expandFrontiers` retrieves all nodes (along with edges) at distance 1 from frontier nodes and returns the set of all rules generated by this one hop expansion. Q_r is therefore initialized with all rules of length 1 starting at x (Line 3). In the main loop, the algorithm checks if the current best rule r is valid or not. If r is valid, it is added to the output and it is not expanded (Line 8). If r is not valid, it is expanded iff the length of its body is less than $maxPathLen$ (Line 10). The termination conditions and the last part of the algorithm are the same of the greedy set-cover algorithm, except that the output may not cover all input examples in G.

To analyze the complexity of Algorithm 1, we assume that each query has a constant cost (linear scan over an index). Each iteration in Algorithm 1 corresponds to the discovery of a rule (valid or invalid), and for each rule we count how many examples from G such a rule covers. The total number of iterations is at most the total number of rules. The worst case is a complete graph where for each predicate p in the KB and for each pair of nodes (x,y), there exists a labelled edge with p that connects x with y. In this case, the number of distinct paths of length $L \leqslant maxPathLen$ between any two nodes of G is $|P|^L$, where $|P|$ is the number of predicates in the KB. The asymptotic complexity of Algorithm 1 is therefore $O(|G| * |P|^L)$, where G is the generation set, and P is the set of predicates in the KB. In reality, most pairs in KBs are connected by very few predicates (1 to 2), thus $|P|$ is small. This is reflected by low execution times for the algorithm in the experiments.

The simultaneous rule generation and selection of Algorithm 1 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 (Line 12). Rather than materializing the entire graph and then traversing it, our solution gradually materializes parts of the graph whenever they are needed for navigation (Lines 3 and 12). Second, the weight estimation prunes unpromising rules. If a rule does not cover new elements in G and does not unbounded cover new elements in V, then it is pruned.

VI. EXPERIMENTS

We implemented the above techniques in RUDIK, our system for Rule Discovery in Knowledge Bases (https://github.com/stefano-ortona/rudik). We carried out an experimental evaluation of our approach and grouped the results in four categories: *(i)* demonstrating the quality of our output for positive and negative rules; *(ii)* comparing our method with the state-of-the-art systems; *(iii)* showing the applicability of rule discovery to create representative training data to learning algorithms; *(iv)* testing the role of the parameters in the system.

Settings. Experiments were run on a desktop with a quad-core i5 CPU at 2.80GHz and 16GB RAM. We used OpenLink Virtuoso, optimized for 8GB RAM, with its SPARQL query endpoint on the same machine. Weight parameters were set to $\alpha = 0.3$ and $\beta = 0.7$ for positive rules, and to $\alpha = 0.4$ and $\beta = 0.6$ for negative rules. We set the maximum number of atoms admissible in the body of a rule ($maxPathLen$) to 3. We discuss the role of these parameters in Section VI-D.

TABLE I. Dataset characteristics.

KB	*Version*	*Size*	*#Triples*	*#Predicates*
DBPedia	3.7	10.06GB	68,364,605	1,424
Yago 3	3.0.2	7.82GB	88,360,244	74
Wikidata	20160229	12.32GB	272,129,814	4,108

Evaluation Metrics. We evaluated the effectiveness in discovering both positive and negative rules. For every KB, we first ordered predicates according to descending popularity (i.e., number of triples having that predicate). We then picked the top 3 predicates for which we knew there existed at least one meaningful rule, and other 2 top predicates for which we did not know whether meaningful rules existed or not.

The evaluation of the discovered rules has been done according to the best practice for rule evaluation [13]. If a rule was clearly semantically correct, we marked all its results over triples as true. If a rule correctness was unknown, we randomly sampled 30 triples either among the new facts (for positive rules) or among the errors (for negative rules), and manually checked them. The *precision* of a rule is then computed as the ratio of correct assertions out of all assertions. While we manually annotated only popular predicates, we executed RuDiK on all predicates in DBPedia and verified that results are consistent even with non popular predicates. Source code and test results, including annotated examples and discovered rules, are available online at https://github.com/stefano-ortona/rudik.

A. *Quality of Rule Discovery in* RuDiK

The first experiment evaluated the accuracy of discovered rules over three KBs: DBPedia, Yago, and Wikidata. Table I shows their characteristics. Over the three KBs, the selected predicates cover 0.2% to 0.4% of the total triples, 0.2% to 8% of the total predicates, 3% to 7% of the total entities, with 8% to 14% entity overlap among the predicates.

Size is important, as loading a KB entirely in memory requires to either use large amount of memory [5], [12], or to shrink it by eliminating the literals [13]. Given the small memory footprint of our algorithm, we can mine rules with commodity HW resources and retain the literals, which are crucial for obtaining expressive rules. While RuDiK takes as input a target predicate at a time, it can discover rules over the entire KB by applying the same procedure on every predicate in it. We discuss next results for subsets of predicates because the manual annotation of the identified new facts and errors is a very expensive process. However, when RuDiK is executed on all the predicates of a KB, results are consistent in terms of number of discovered rules and execution times. For example, for 600 predicates in DBPedia we mined about 3000 positive rules, with at most 26 rules for a predicate, and 4000 negative rules, with at most 32 rules for a predicate.

Positive Rules RuDiK. We evaluate the precision for the positive discovered rules on the top 5 predicates for each KB. The number of new induced facts varies significantly from rule to rule. To avoid the overall precision to be dominated by such rules, we first compute the precision for each rule, and then average values over all induced rules. Table II reports precision values, along with predicates average running time, and the number of manually annotated triples. We distinguish predicates for which we knew there existed at least one correct rule (in bold), and all predicates (in brackets).

TABLE II. RuDiK Positive Rules Accuracy.

KB	*Avg. RunTime*	*Avg. Precision over Predicates with Rules (All)*	*# Labeled Triples*
DBPedia	35min	**87.86**% (63.99%)	139
Yago 3	59min	**79.17**% (62.86%)	150
Wikidata	141min	**85.71**% (73.33%)	180

As precision varies across different KBs and facts, we report the value for every predicate. For DBPedia: academicAdvisor (100%), child (58%), spouse (97%), founder (no valid rules), successor (68%). Yago: hasChild (50%), influences (35%), isLeaderOf (70%), isMarriedTo (100%), exports (83%). Wikidata: spouse (100%), child (76%), paintingCreator (60%), academicAdvisor (100%), subsidiary (67%). Average precision values are brought down by few predicates, such as `founder`, where meaningful positive rules probably do not exist at all. Our experience show that it suffices to read the rules to recognize that they are semantically wrong and should be discarded, e.g., a human immediately sees that it is not possible to derive a founder from the KB's predicates.

The running time is influenced by the size of the KB. The more edges we have on average for a node (entity), the more alternative paths we test while traversing the graph. Another relevant aspect is the target predicate involved. Some entities have a large number of outgoing and incoming edges, e.g., entity "*United States*" in Wikidata has more than 600K. When the generation set includes such entities, the navigation of the graph is slower. Parameter $maxPathLen$ also impacts the running time. The longer the rule, the bigger is the search space, as we discuss in Section VI-D.

TABLE III. RuDiK Negative Rules Accuracy.

KB	*Avg. Run Time*	*# Pot. Errors*	*Precision*
DBPedia	19min	499 (84)	**92.38**%
Yago 3	10min	2,237 (90)	**90.61**%
Wikidata	65min	1,776 (105)	**73.99**%

Negative Rules RuDiK. We evaluate discovered negative rules as the percentage of correct errors identified for the top 5 predicates in each KB. Table III shows, for each KB, the total number of potential erroneous triples found with the discovered rules, whereas the precision is computed as the percentage of actual errors among potential errors. Numbers in brackets show the number of triples manually annotated to obtain the precision. At the predicate level, the results are the following. DBPedia: academicAdvisor (29%), child (90%), spouse (87%), founder (95%), ceremonialCounty (100%). Yago: hasChild (82%), isMarriedTo (97%), created (100%), hasAcademicAdvisor (100%), wroteMusicFor (43%). Wikidata: spouse (78%), child (82%), founder (100%), creator (48%), oathGiven (100%).

Negative rules have better accuracy than positive ones when considering all predicates. This is due to the fact that negative rules exist more often than positive rules. While quality of the rules is good, especially on the more noisy KBs, we also discover rules that are supported by the large majority of the data, but do not hold semantically. For example, we identify the rule that two people with same gender cannot be married both in Yago and Wikidata. Such rule has a 94% precision in Yago and 57% in Wikidata. Differently from positive rules, literals play a key role in negative rules. In fact, several correct negative rules rely on temporal aspects in which something cannot happen before/after something else.

TABLE IV. AMIE Dataset characteristics.

KB	*Size*	*#Triples*	*#Predicates*	*#*`rdf:type`
DBPedia	551M	7M	10,342	22.2M
Yago 2	48M	948.3K	38	77.9M

Temporal information is usually expressed through dates and years, which are represented as literal values in KBs.

Discovering negative rules is faster than discovering positive rules because of the different nature of the examples covered by validation queries. Whenever we identify a candidate rule, we execute the body of the rule against the KB with a SPARQL query to compute its coverage over the validation set. These queries are faster for negative rules since the validation set only contains entities directly connected by the target predicate, whereas in the positive case the validation set corresponds to counter examples that do not have this property and are more expensive to evaluate.

For non popular predicates, the system found rules with quality comparable to the popular predicates. For example, it discovers the valid negative rule $\texttt{routeStart}(x,a) \wedge \texttt{routeEnd}(x,b) \Rightarrow \texttt{notMeetingRoad}(a,b)$ for predicate `meetingRoad` with just 114 facts in DBPedia, and the valid positive rule $\texttt{highestState}(a,x) \wedge \texttt{municipality}(b,x) \Rightarrow \texttt{highestRegion}(a,b)$ for predicate `highestRegion` with just 36 facts.

B. Comparative Evaluation

We compared our methods against AMIE [13], a state-of-the-art positive rule discovery system for KBs. AMIE assumes that the given KB fits into memory and discovers positive rules for every predicate. It then outputs all rules that exceed a given threshold and ranks them according to a coverage function.

Given its in-memory implementation, AMIE went out of memory for the KBs of Table I on our machine. Thus, we used the modified versions of Yago and DBPedia from the AMIE paper [13], which are devoid of literals and `rdf:type` facts. Removing literals and `rdf:type` triples drastically reduce the size of the KB. Since our approach needs type information (for the generation of G and V and for the discovery of inequality atoms), we run AMIE on its original datasets, while for our algorithm we used the AMIE dataset plus `rdf:type` triples. Last column of Table IV reports the number of triples added to the original AMIE dataset.

Positive Rules Comparison. For this experiment we ran RuDiK as follows: we first list all the predicates in the KB that connect a *subject* to an *object*. We then computed for both subject and object the most popular `rdf:type` that is not super class of any other most popular type. We finally ran our approach sequentially on every predicate, with $maxPathLen = 2$ (AMIE default setting). AMIE discovers 75 output rules in Yago, and 6090 in DBPedia. We followed their experimental setting and picked the first 30 best rules according to their score. We then picked the rules produced by our approach on the same head predicate of the 30 best rules output of AMIE.

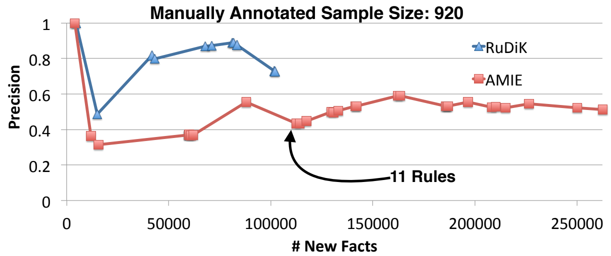

Fig. 3. Accuracy for new facts identified by executing rules in descending AMIE's score on Yago 2 (no literals).

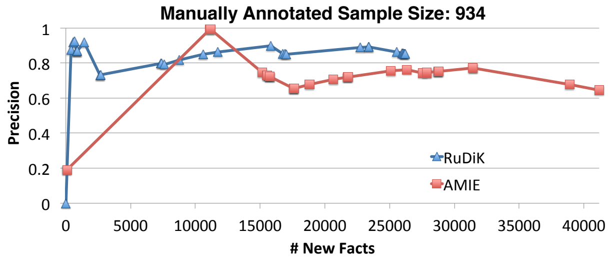

Fig. 4. Accuracy for new facts identified by executing rules in descending score on DBPedia (no literals).

Figures 3 and 4 report the results on Yago and DBPedia, respectively. We plot the total cumulative number of new unique facts (x-axis) versus the aggregated precision (y-axis) when incrementally including in the solution the rules according to their descending (AMIE's) score. Rules from AMIE produce more predictions, but with significant lower accuracy in both KBs. This is because many good rules are preceded by meaningless ones in the ranking, and it is not clear how to set a proper k to get the best ones. In RuDiK, instead of the conventional ranking mechanism, we use a scoring function that discovers only inherently meaningful rules with enough support. As a consequence, RuDiK outputs just 11 rules for 8 target predicates on the entire Yago – for the remaining predicates RuDiK does not find any rule with enough support. If we limit the output of AMIE to the best 11 rules in Yago (same output as our approach), its final accuracy is still 29% below our approach, with just 10K more predictions.

Negative Rules Comparison. While AMIE has not been designed to discover negative rules, we created a baseline solution on top of it. First, we created a set of negative examples (Section IV-B) for each predicate in the top-5. For each example, we added a new fact to the KB by connecting the two entities with the *negation* of the predicate. For example, we added a `notSpouse` predicate connecting each pair of people who are not married according to our generation technique. We then ran AMIE on these new predicates.

Table V shows that RuDiK outperforms AMIE in both cases with an absolute precision gain of almost 20% (41-49% relative). The drop in quality for RuDiK w.r.t. the results in Section VI-A is because of the KBs without literals. Numbers in brackets show the number of triples manually annotated.

TABLE V. Negative Rules vs AMIE.

	AMIE		RuDiK (no literals)	
KB	*# Errors*	*Precision*	*# Errors*	*Precision*
DBPedia	457 (157)	38.85%	148 (73)	**57.76%**
Yago 2	633 (100)	48.81%	550 (35)	**68.73%**

Running Time. On our machine, AMIE could finish the computation on Yago 2, while for other KBs it got stuck after some time. For these cases, we stopped the computation if there were no changes in the output for more than 2 hours. Running times for AMIE are different from [13], where it was run on a 48GB RAM server.

TABLE VI. TOTAL RUN TIME COMPARISON.

KB	*#Predicates*	*AMIE*	RUDIK	*Types*
YAGO 2	20	30s	18m,15s	12s
YAGO 2s	26 (38)	> 8h	47m,10s	11s
DBPEDIA 2.0	904 (10342)	> 10h	7h,12m	77s
DBPEDIA 3.8	237 (649)	> 15h	8h,10m	37s
WIKIDATA	118 (430)	> 25h	8h,2m	11s
YAGO 3	72	-	2h,35m	128s

Table VI reports the running time on different KBs. The first five KBs are AMIE modified versions, while YAGO 3 includes literals and `rdf:type`. The second column shows the total number of predicates for which AMIE produced at least one rule before getting stuck, while in brackets we report the total number of predicates in the KB. The third and fourth columns report the total running time of the two approaches. Despite being disk-based, RUDIK successfully completes the task faster than AMIE in all cases, except for YAGO 2. This is because of the very small size of this KB, which fits in memory. However, when we deal with complete KBs (YAGO 3), the KB could not even be loaded due to out of memory errors. The last column reports the running time to compute `rdf:type` information for all predicates.

Other Systems. In [2], the system mines rules that are less general than our approach; on YAGO 2, it discovers 2K new facts with a precision lower than 70%, while our rule on YAGO 2 already produces more than 4K facts with a 100% precision. Another system [5] implements AMIE algorithm with a focus on scalability and the output is the same as AMIE. We did not compare with Inductive Logic Programming systems [8], [23], as these are already significantly outperformed by AMIE both in accuracy and running time.

C. Machine Learning Application

The goal of this experiment is to test RUDIK's ability in providing valid training examples to ML models. We chose DeepDive [19], a framework for information extraction. DeepDive extracts entities and relations from text articles via distant supervision. The key idea in distant supervision is to use an external source of information (e.g., a KB) to provide training examples for a supervised algorithm. For example, DeepDive can extract mentions of married couples from text documents. In this scenario, it uses a KB to label pairs of married couples that can be found in DBPEDIA as *true* positive example. As KBs provide facts, in DeepDive the burden of creating negative examples is left to the user. We compare the output of DeepDive upon its spouse example trained with different sets of negative examples over two datasets.

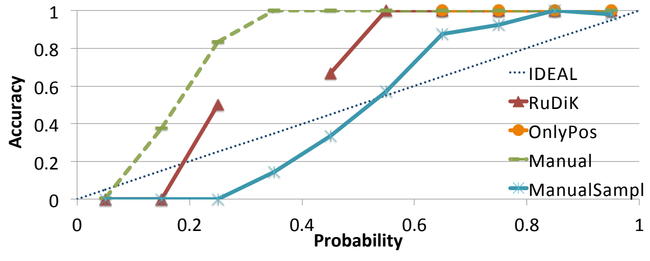

Fig. 5. DeepDive executions with different training examples – 1K articles.

Figure 5 shows DeepDive accuracy plot with 1K input documents. The plot shows the fraction of correct positive predictions over total predictions (y-axis), for each output probability value (x-axis). The ideal execution, marked by the dotted blue line, would predict all facts with a probability of 1 and zero facts with an output probability of 0. The best algorithm deflects the least from the blue dotted line, and this distance is our evaluation metric. RUDIK is the output of DeepDive using our discovered rules to generate negative examples on DBPEDIA. `OnlyPos` uses only positive examples from DBPEDIA, `Manual` uses positive examples from DBPEDIA and manually defined rules to generate negative examples, while `ManualSampl` uses a sample of the manually generated negative examples in size equal to positive examples. `OnlyPos` and `Manual` do not provide valid training, as the former has only positive examples and labels everything as true, while the latter has many more negative examples than positive ones and labels everything as false. `ManualSampl` is the winner, while our approach suffers from the absence of data to mine: over the input 1K articles, there are only 20 positive examples from DBPEDIA. The lack of evidence in the training data also explains the missing points for RUDIK, with no prediction in the probability range 25-45%.

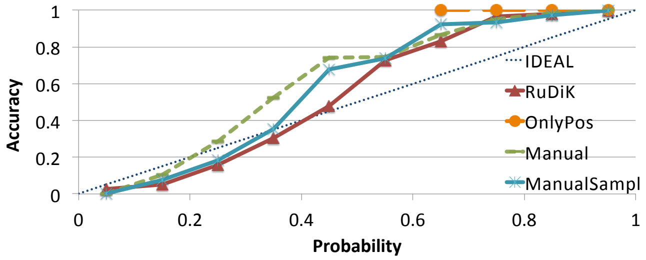

Fig. 6. DeepDive executions with different training examples – 1M articles.

When we extend the input to 1M articles, things change drastically (Figure 6). All approaches except `OnlyPos` successfully drive the training, with the examples provided with RUDIK leading to the best result. This is because of the quality of the negative examples: our rules generate representative examples that are correct (thanks to the LCWA), semantically related (thanks to the constraint on the predicate connecting them), and have the number of negative examples in the same order of magnitude of the positive ones. The correct and rich examples enable DeepDive to identify discriminatory features between positive and negative labels. The output of `ManualSampl` and RUDIK are very similar, meaning that we can use our approach to simulate user behavior and automatically produce negative examples.

D. Internal Evaluation

We outline the impact of individual components in RUDIK. Full results are reported in the technical report online at http://www.eurecom.fr/publication/5321.

KB Noise Impact. In terms of quality of the KBs, the percentages of erroneous triples identified by our rules are 0.23% for WIKIDATA, 0.26% for DBPEDIA, and 0.6% for YAGO. To study the impact of errors in the KB, we first manually removed errors from the top five predicates in DBPEDIA to obtain clean positive and negative examples. We collected such rules and consider them the best possible output. We then gradually introduced errors by switching positive and negative examples between their sets. Figure 7 shows the accuracy degradation

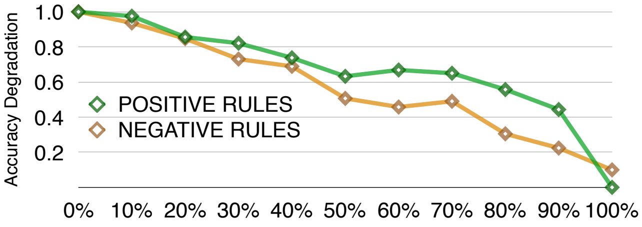

Fig. 7. KB Noise Impact on Rules Quality.

averaged over predicates (y-axis) from 0% errors to 100% (x-axis). As expected, the accuracy decreases with the amount of errors. RuDiK is robust enough to deliver mostly correct rules until 40% of errors, while after that accuracy starts to drop significantly. An interesting point is that even with 90% of errors, RuDiK is still able to isolate the 10% of good examples to mine at least one valid rule.

LCWA. We study the effect of the LCWA assumption for the generation of negative examples. Given a predicate p, we tested three generation strategies: RuDiK strategy (Section IV-B), 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 but x and y do not have to be connected by a predicate different from p). Table VII reports quality results for the discovered rules. *Random* and *LCWA* show similar behavior, with a slightly better precision than RuDiK. This is because by randomly picking examples from the Cartesian product of subject and object, the likelihood of getting entities from different time periods is very high, and negative rules pivoting on time constraints are usually correct. Instead, by forcing x and y to be connected with different predicate, we generate semantically related examples that lead to more rules. Rules such as `parent`$(a, b) \Rightarrow$ `notSpouse`(a, b) are not generated with random strategies, since the likelihood of picking two people that are in a parent relation is very low. The RuDiK strategy enables the discovery of more types of rules, and not only rules involving time constraints.

TABLE VII. Impact of Examples Generation on DBPedia.

Strategy	*# Potential Errors*	*Precision*
Random	247	**95.95%**
LCWA	263	95.82%
RuDiK	**499**	92.38%

Effect of Literals. Table VIII reports the output precision obtained by enabling and disabling the use of literal comparisons in RuDiK. Including literal values has a considerable impact on accuracy, both for positive and negative rules. Negative rules without literals find less than half potential errors (numbers in brackets) with lower precision. For predicate `founder`, RuDiK discovers 79 potential errors with a 95% precision with literal rules, while none are detected by using rules without literals. Interestingly, including literals reduces also the running time. This is due to the pruning effect of the A^* search, literals enable the early discovery of good rules.

Rule Length Impact. The $maxPathLen$ parameter fixes the maximum number of atoms in the body of a rule. Low values may exclude from the search space meaningful rules, while high values exponentially increase the search space and consequently the running time. With $maxPathLen = 2$, there is a significant improvement in running time, but meaningful rules are lost and precision drop to 49% for positive rules and 90% for negative ones. In particular, we lose rules with literals comparison, as these require at least three atoms in the body. At the other side of the spectrum, with $maxPathLen = 4$ the search space explodes and RuDiK could not finish the computation within 24 hours for any predicate. We measured the accuracy of rules discovered in 24 hours of computation and the results are comparable to those computed with $maxPathLen = 3$, with a small increase in precision for positive rules and a small drop for negative ones. Rules with length 4 are more complex to understand, and when executed over the KB they often return an empty result because of their higher selectivity. We therefore set $maxPathLen = 3$ as a compromise between efficiency and accuracy.

TABLE VIII. Impact of Literals on DBPedia.

	With Literals		**Without Literals**	
Rules	*Run Time*	*Precision*	*Run Time*	*Precision*
Pos.	~35min	**63.99%**	~54min	60.49%
Neg.	~19min	**92.38%** (**499**)	~25min	84.85% (235)

Weight Parameters. For positive rules, the best assignment is $\alpha = 0.3$ and $\beta = 0.7$, while for negative rules is $\alpha = 0.4$ and $\beta = 0.6$. Since discovering correct positive rules is more challenging than negative ones, favoring precision over recall gives the best accuracy, while for negative rules we can be more recall oriented. In both positive and negative settings, the variation in performance for $\alpha \in [0.1, 0.9]$ is limited ($\leqslant 12\%$), showing the robustness of the set cover problem formulation.

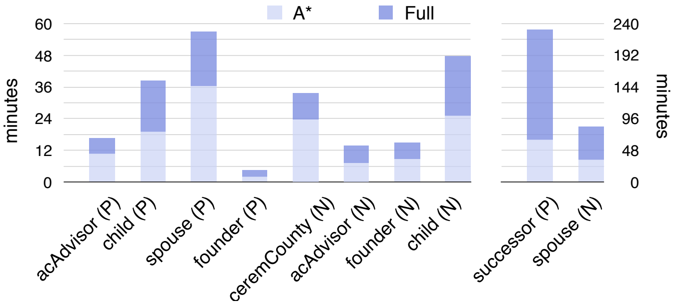

Fig. 8. A^* Pruning Runtime Improvement.

Search. We quantify the benefit of the A^* algorithm on the running time. Figure 8 shows the running time, for each predicate, of the A^* algorithm (light-colored bars) against a modified version that first generates the universe of all possible rules, and then applies the greedy set cover algorithm on such a universe (dark-colored bars). The last two predicates refer to the y-axis labels on the right hand side, as they have higher running times. In the figure, `(P)` indicates positive rules and `(N)` negative ones. The A^* strategy shows an average 50% improvement in running times as it avoids the generation of unpromising paths and the loading of the corresponding RDF instances from disk. When there exist rules that cover many examples from the generation set (e.g., `successor (P)`, `founder (P)`), the algorithm identifies such rules rather early, thus pruning several unpromising paths. In such cases the running time improvement is above 70%.

Set Cover. Our set cover 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. Oftentimes correct rules are not among the top-10 ranked, and we found cases where 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 standard variant of RuDiK.

VII. Related Work

A significant body of work has addressed the problem of discovering constraints over *relational data*, e.g., [6]. However, these techniques cannot be applied to KBs because of the schema-less nature of RDF data and the OWA. Traditional approaches rely on the assumption that data is either clean or has a negligible amount of errors, which is not the case with KBs, and, even when the algorithms are designed to tolerate errors [1], [15], a direct application of relational database techniques on RDF KBs requires the prohibitive materialization of all possible predicate combinations into relational tables. Recently, theoretical foundations of Functional Dependencies on Graphs have been laid [11]. However, their language covers only a portion of our negative rules and does not include general literal comparisons.

Rule mining approaches designed for positive rule discovery in RDF KBs load the entire KB into memory prior to the graph traversal step [13], [5]. This is a limitation for their applicability over large KBs, and neither of these two approaches consider value comparison. In contrast to them, RuDiK load in memory a small fraction of the KB. This makes it scalable and the low memory footprint enables a bigger search space with rules that have literal comparisons. Finally, association rules can be mined to recommend new facts [2], but such rules are made of constants only and are therefore less general than the rules generated by RuDiK.

ILP systems such as WARMR [8], Sherlock [18], and ALEPH[1] are designed to work under the CWA and require the definition of positive and negative error-free examples. It has been showed how this assumption does not hold in KBs and that AMIE outperforms this kind of systems [13]. Detection of semantic errors in KBs has also been tackled with approaches that are orthogonal to negative rules. For example, discovering domain and range restrictions [23], or identifying outliers after grouping subjects by type [25]. Finally, the output of our rules can be modeled as the result of a link prediction problem over the KB [10]. However, we focus on logical rules for their benefits as "white boxes", including the possibility of doing static analysis, execution optimization, and interpretability.

VIII. Conclusion

We presented RuDiK, a rule discovery system that mines both positive and negative rules on noisy and incomplete KBs. Positive rules identify new valid facts for the KB, while negative rules identify errors. We experimentally showed that our approach generates concise sets of meaningful rules with high precision, is scalable, and can work with exisisting KBs.

Open questions are related to the interactive discovery of the rules, if and how it is possible to drastically reduce the runtime of the discovery without compromising the quality of the rules. Another interesting direction is to discover more expressive rules that exploit temporal information through smarter analysis of literals [1], e.g., "if two person have age difference greater than 100 years, then they cannot be married".

Acknowledgement. This work was partly supported by Google with a Faculty Research Award.

References

[1] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. Temporal rules discovery for web data cleaning. *PVLDB*, 9(4):336–347, 2015.

[2] Z. Abedjan and F. Naumann. Amending RDF entities with new facts. In *ESWC*, pages 131–143, 2014.

[3] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia-A crystallization point for the web of data. *J. Web Semantics*, 7(3):154–165, 2009.

[4] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pages 1247–1250, 2008.

[5] Y. Chen, S. Goldberg, D. Z. Wang, and S. S. Johri. Ontological pathfinding. In *SIGMOD*, pages 835–846, 2016.

[6] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[7] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[8] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data mining and knowledge discovery*, 3(1):7–36, 1999.

[9] O. Deshpande, D. S. Lamba, M. Tourn, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Building, maintaining, and using knowledge bases: a report from the trenches. In *SIGMOD*, pages 1209–1220, 2013.

[10] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *KDD*, 2014.

[11] W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In *SIGMOD*, pages 1843–1857, 2016.

[12] M. H. Farid, A. Roatis, I. F. Ilyas, H. Hoffmann, and X. Chu. CLAMS: bringing quality to data lakes. In *SIGMOD*, pages 2089–2092, 2016.

[13] 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.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[15] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *TCS*, 149(1):129–149, 1995.

[16] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

[17] J. Pujara, H. Miao, L. Getoor, and W. Cohen. Knowledge graph identification. In *ISWC*, pages 542–557, 2013.

[18] S. Schoenmackers, O. Etzioni, D. S. Weld, and J. Davis. Learning first-order horn clauses from web text. In *EMNLP*, 2010.

[19] 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.

[20] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A core of semantic knowledge unifying Wordnet and Wikipedia. In *WWW*, 2007.

[21] F. M. Suchanek, M. Sozio, and G. Weikum. SOFIE: A self-organizing framework for information extraction. In *WWW*, pages 631–640, 2009.

[22] P. Suganthan GC, C. Sun, H. Zhang, F. Yang, N. Rampalli, S. Prasad, E. Arcaute, G. Krishnan, et al. Why big data industrial systems need rules and what we can do about it. In *SIGMOD*, pages 265–276, 2015.

[23] G. Töpper, M. Knuth, and H. Sack. Dbpedia ontology enrichment for inconsistency detection. In *I-SEMANTICS*, pages 33–40, 2012.

[24] D. Vrandečić and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[25] D. Wienand and H. Paulheim. Detecting incorrect numerical data in dbpedia. In *ESWC*, 2014.

[1] https://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph