Robust Discovery of Positive and Negative Rules in Knowledge-Bases

ABSTRACT

We present RuDiK, a system for the discovery of declarative rules over knowledge-bases (KBs). RuDiK discovers rules that rely on *positive* relationships between entities, such as "if two persons have the same parent, they are siblings", and *negative rules*, i.e., patterns that lead to 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 detecting erroneous triples. 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)* allow the discovery of *approximate* rules to be robust to errors and incompleteness in the KB, *(iii)* use disk-based algorithms, effectively enabling the mining of large KBs in commodity machines. 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.

ACM Reference format:
. 2016. Robust Discovery of Positive and Negative Rules in Knowledge-Bases. In *Proceedings of ACM Conference, YYYY, XXXX 2017 (Conference'17),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 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*, or relation, expresses a binary relation between a *subject* and a *object*. KB triples, called facts, store information about real-world entities and their relationships, such as "Michelle Obama is married to Barack Obama", or "Larry Page is the founder of Google". Significant effort has been put on KBs creation in the last 10 years in the research community (DBPedia [6], FreeBase [7], Wikidata [28], DeepDive [25], Yago [26], TextRunner [5]) as well as in the industry (e.g., Google [13], Wal-Mart [12]).

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 problems. First, false facts in the KB are propagated from the sources, or introduced by the extractors. 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 not hold [13, 16], we cannot assume that a missing fact is false (*open world assumption*).

As a consequence, the amount of errors and incompleteness in KBs is usually significant, with up to 30% errors for facts derived from the Web [1, 27]. 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 natural approach to assist curators of KBs is to discover *declarative rules* that can be executed over the KB to improve the quality of the data [2, 8, 16]. We target the discovery of two different 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 1.1. Consider a KB with information about parent and child relationships. A positive rule r_1 can be:

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

It states that if a person a is parent of person b, then b is child of a. A negative rule r_2 has similar form, but different semantics (*DOB* stands for Date Of Birth):

$$\texttt{DOB}(a, v_0) \wedge \texttt{DOB}(b, v_i) \wedge v_0 > v_i \wedge \texttt{child}(a, b) \Rightarrow \mathit{false}$$

It states that person b cannot be child of a if a was born after b. By instantiating the rule for `child` facts, we identify erroneous triples stating that a child is born before a parent.

While intuitive to humans, the rules above must be manually stated in a KB to be enforced, and there are thousands of rules in a large KB [17]. A rule discovery system should therefore discover both positive and negative rules. However, three main challenges arise when discovering rules for KBs.

Errors. Traditional techniques for rule discovery assume that data is either clean or has a negligible amount of errors [3, 9, 20]. We will show that KBs present a high percentage of errors and we need techniques that are noise tolerant.

Open World Assumption. Other approaches rely on the presence of positive and negative examples [11, 23], 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, 8, 15, 16]. Given the large and increasing size of KBs, these approaches aggressively prune the search space by focusing on a simple language.

We present RuDiK (Rule Discovery in Knowledge Bases), a novel system for the robust discovery of positive and negative rules over KBs that addresses these challenges. RuDiK is the

Conference'17, YYYY

DOI: 10.1145/nnnnnnn.nnnnnnn

first system capable of discovering both *positive and negative rules* over noisy and incomplete KBs. Moreover, by relying on disk based algorithms, RuDiK discovers rules with a richer language that allows value comparisons. This increase in the *expressive power* enables a larger number of patterns that can be expressed in the rules, and therefore a larger number of new facts and errors that can be identified. These results are achieved by exploiting the following contributions.

Problem Definition. We formally define the problem of rule discovery over erroneous and incomplete KBs. In contrast to the traditional ranking of a large set of rules based on a measure of support [11, 16, 24], our problem definition aims at the identification of a compact subset of approximate rules. Given errors in the data and incompleteness, the ideal solution is a compact set of rules that cover the majority of the positive examples, and as few negative examples as possible. We map the problem to the weighted set cover problem (Section 3).

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 examples is a tedious exercise that requires manual work. We design an algorithm for example generation that is aware of the errors and inconsistency in the KB. Our generated examples lead to better rules than examples obtained with alternative approaches (Section 4).

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 its paths. By materializing only the portion of the KB that is needed for the discovery, the disk is accessed only whenever the navigation of a specific path is required (Section 5).

We experimentally RuDiK on three popular and widely used KBs, namely DBPedia [6], Yago [26], and Wikidata [28] (Section 6). 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, RuDiK performs consistently well with KBs of all sizes, while other systems cannot scale or fail altogether.

2 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 and consistency, 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.

2.1 Language

Our goal is to automatically discover *Horn Rules* in KBs. A Horn Rule is a disjunction of *atoms* with at most one unnegated atom. When written in the implication form, Horn Rules have one of the following formats:

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

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). The head of the rule is either unnegated (left) or empty (right). We call the former *definite clause* or *positive rule*, as it generates new facts (e.g., r_1 in Example 1.1). We call the latter *goal clause* or *negative rule* (e.g., r_2 in Example 1.1), as it identifies incorrect facts, similarly to denial constraints for relational data [9]. An atom is a predicate connecting two variables, two entities, or an entity and a variable. For simplicity, we represent an atom with the notation $\texttt{rel}(a, b)$, where $\texttt{rel}$ is a 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. We define the variables in the head of the rule as the *target variables*. We also write negative rules as definite clauses by rewriting a body atom in the head. The result is a logically equivalent formula that emphasizes the generation of negative facts. In Example 1.1, r_1 is a positive rule, where new positive facts are identified with target variables a and b. Rule r_2 is a negative rule to identify errors; we can rewrite it as a definite clause to derive false facts from the KB and obtain 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 example, 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 $\neq$. In such a case a and b can be also assigned to entities. We will explain later on why this exception is important.

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 in r_{body} can be instantiated.

As in other approaches [8, 16], we define a rule *valid* iff it satisfies the following constraints.

Connectivity. An atom A_1 can be reached by an atom A_2 iff A_1 and A_2 share at least one variable or one entity. In a valid rule, every atom in must be *transitively* reachable by any other atom in the rule.

Repetition. Every variable in a rule must appear at least twice. Since target variables already appear once in the head of the rule, each variable that is not a target variable must be involved in a join or in a literal comparison.

2.2 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 2.1. We denote with E the set of all possible pairs of entities in kb. The coverage of r'_2 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 (or both), 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 affirm that r'_2 does not cover (x, y). Given that KBs are largely incomplete [22], discriminating between missing and conflicting information is important. 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:

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

3 ROBUST RULE DISCOVERY

We define the discovery for a *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 target predicate. Consider the discovery of positive rules for the child predicate; G contains examples of real pairs of parents and children and V contains pairs of people who *are not* in a child relation. If we want to identify errors in the child predicate, 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 real pairs. We explain in Section 4.2 how to generate these two sets for a given predicate.

We now formalize the *exact discovery problem.*

Definition 3.1. Given a KB kb, two sets of pairs of entities G and V from kb with $G \cap V = \varnothing$, and all the valid Horn Rules R for kb, a solution for the *exact discovery problem* is a subset R' of R s.t.:

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

The exact solution is a 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 very little impact when applied on the KB. In fact, given a pair of entities (x, y), there is always an overfitting rule whose body covers them by assigning target variables to x and y as shown next.

Example 3.2. 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 positive rules:

$$R_1 : \texttt{livesIn}(a, v_0) \wedge \texttt{livesIn}(b, v_0) \Rightarrow \texttt{couple}(a, b)$$

$$R_2 : \texttt{hasChild}(a, v_i) \wedge \texttt{hasChild}(b, v_i) \Rightarrow \texttt{couple}(a, b)$$

$$R_3 : \texttt{hasChild}(Michelle, Malia) \wedge \texttt{hasChild}(Barack, Malia) \Rightarrow \texttt{couple}(Michelle, Barack)$$

Rule R_1 states that two persons are a couple if the live in the same place, while rule R_2 states that they are a couple if they have a child in common. Both rules cover the positive example, but R_1 covers also the negative one. R_2 is an exact solution. Rule R_3 explicitly mentions entity values in its head and body. It is also an exact solution, but overfits the given positive example.

The example highlights that the exact discovery problem is not robust to data quality 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, e.g., the rule misses a good example because a child relation is missing for M. Obama. 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 very little effect on the KB.

3.1 Weight Function

Given errors and missing information in both G and V, we drop the requirement of exactly covering the sets with the

rules. However, coverage is a strong indicator of quality: good rules should cover several examples in G, while covering several elements in V is an indication of incorrect rules. We model these ideas as a *weight* associated with every rule.

Definition 3.3. Given a KB kb, two sets of pair of entities G and V from kb with $G \cap V = \emptyset$, and a Horn Rule r, the *weight of* r is defined as follow:

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

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 the two 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 that have the information tested by the rule's body.

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 as the discovered rules cover as many generation examples as possible.

Example 3.4. Consider again rule r_2' 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 in kb.

Definition 3.5. Given a set of rules R, the *weight for* R is:

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

Weights enable the modeling of the presence of errors in KBs. We will show in the experimental evaluation that several semantically correct rules have a significant coverage over V, which corresponds to errors in the KB.

3.2 Problem Definition

We can now state the approximate version of the problem.

Definition 3.6. Given a KB kb, two sets of pair of entities G and V from kb with $G \cap V = \emptyset$, all the valid Horn Rules R for kb, and a w weight function for R, a solution for the *approximate discovery problem* is a subset R' of R such that:

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

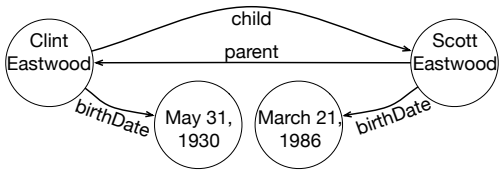

Figure 1: Graph example of DBPedia for four triples.

The approximate 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.

We can map this problem to the *weighted set cover problem*, which is proven to be NP–complete [10]. The reduction follows immediately from the following mapping: the set of elements (universe) corresponds to the generation examples in G, the input sets are identify by the rules defined in R, the non-negative weight function $w : r \rightarrow \mathbb{R}$ is $w(r)$ in Definition 3.3, and the cost of R is defined to be its total weight, according to Definition 3.5.

4 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.

In the following we discuss the discovery of positive rules, i.e., 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 negative examples (false facts), while the validation set V is built from the set of positive examples (true facts).

4.1 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. The smaller the size of G, the smaller is the search space for rule generation.

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 a portion of DBPedia [6] for four triples.

The body of a rule can be seen as a path in the graph. In Figure 1, the body $\texttt{child}(a, b) \wedge \texttt{parent}(b, a)$ corresponds to the path *Clint Eastwood* → *Scott Eastwood* → *Clint Eastwood*. As introduced in Section 2.1, a valid body of a rule 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 from x to y (and vice versa) 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 while 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 constrains the search space by determining the maximum number of edges in the path. When translating paths to Horn Rules, $maxPathLen$ is the maximum number of atoms allowed in the body of the rule. We will validate the use of this parameter in Section 6. 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 (validation queries).

One of the goals is to discover more expressive rules. We therefore generate three new atom types as detailed next.

Literal comparison. In Section 2.1 we defined our target language, which, other than predicate atoms, includes literal comparison. Their role is to enrich the language with comparisons among literal values other than equalities, such as "greater than". 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 should contain an edge '$<$' from node "*March 31, 1930*" to node "*March 21, 1986*". Unfortunately, the original KB does not contain this kind of information, and materializing such edges among all literals is infeasible.

The generation set G is the resource point to identify relevant literal comparison. Since we discover paths for a pair of entities from G in isolation, the size of a graph for a pair of entities is relatively small, thus we can afford to compare all literal values within a single example graph. KBs include three types of literals: numbers, dates, and strings. 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:

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

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 $\texttt{rdf:type}$ triples associated to entities. We add an inequality edge in the input example graph only between those pairs of entities of the same type (as in the president example above).

Constants. Finally, we allow the discovery of rules with constant selections. Suppose that for the above rule for president, all examples in G are people born in the country "*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:

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

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.

4.2 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 possible simple way of creating false facts is to randomly select pairs from the Cartesian product of the entities [23]. While this process gives negative examples with a very high precision, only a very small fraction of these entity pairs are semantically related (something that is guaranteed for positive examples since pairs in G are always connected

at least by the target predicate). This semantic aspect has strong effects in the ultimate 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.

To generate negative examples that are likely to be correct (true false facts) and that are semantically related in the KB, we mine the facts to identify the entities that are more likely to be complete (i.e., entities for which the KB contains every piece of information). This process is done exploiting and extending the popular notion of *Local-Closed World Assumption* (*LCWA*) [13, 16]. 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 extend this intuition also to predicates rather than entities. If a KB contains a relation between two entities x and y, then it contains all relations between x and y.

Now that we can identify entities that are likely to be complete, 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. Moreover, for a candidate negative example over entities (x, y), x must be connected to y via a predicate that is different from the target predicate. In other words, given a KB kb and a target predicate rel, (x, y) is a negative examples if $\langle x, rel', y \rangle \in kb$, with $rel' \neq rel$. These restrictions make the size of V of the same order of magnitude of G (see Section 6), and guarantees that, for every $(x, y) \in V$, x and y are related by at least one predicate.

Example 4.1. 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) in either G or V, x and y are always of the same *type*. Note that even when the LCWA does not hold, the generated negative examples are still useful for the discovery of rules. We experimentally validate these observations in Section 6.3.

5 DISCOVERY ALGORITHM

We introduce a greedy approach to solve the approximate version of the discovery problem (Section 3.2). The algorithm combines two phases: *(i)* it solves the set cover problem with a greedy strategy; *(ii)* it discovers new rules by navigating the graph in the A^* search fashion.

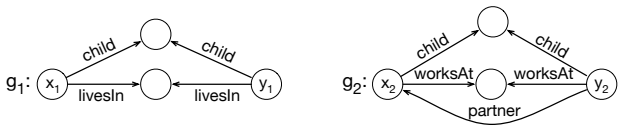

Figure 2: Two positive examples.

5.1 Marginal Weight

We follow the intuitions behind the greedy algorithm for weighted set cover by defining a *marginal weight* for rules that are not yet included in the solution [10].

Definition 5.1. 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 total weight increase of adding r to an existing set of rules R. In other words, it indicates the contribution of r to R in terms of new elements covered in G and new elements unbounded covered in V. Since the set cover problem aims at minimizing the total weight, we would never add a rule to the solution if its marginal weight is greater than or equal to 0.

The algorithm for greedy rule selection is then straightforward: given a generation set G, a validation set V, and the universe of all possible rules R, the algorithm picks at each iteration the rule r with minimum marginal weight and add it to the solution R_{opt}. 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_{opt} 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 marginal weight is greater than or equal to 0 whenever (i) the rule does not cover new elements in G, and (ii) it does not unbounded cover new elements in V. If the second termination condition is not met, there may exist examples in G that are not covered by R_{opt}. In such a case the algorithm will augment R_{opt} with single-instance rules (rules that cover only one example), one for each element in G not covered by R_{opt}.

The greedy solution guarantees a `log(k)` approximation to the optimal solution [10], where k is the largest number of elements covered in G by a rule in R and k is at most $|G|$. If the optimal solution is made of rules that cover disjoint sets over G, then the greedy solution coincides with the optimal one.

5.2 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 really need all possible paths for every example in G?

Example 5.2. Consider the scenario where we are mining positive rules for the target predicate `spouse`. The generation set G includes two examples g_1 and g_2, Figure 2 shows the corresponding KB graphs. 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. Looking at the KB 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 become redundant.

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. The same intuition is behind the A^* graph traversal algorithm [19]. Given an input weighted graph, A^* computes the smallest cost path from a starting node s to an ending node t. At each iteration, A^* maintains a queue of partial paths starting from s, and it expands one of these paths based on an *estimation* of the cost to reach t. The path with the best estimation is expanded and added to the paths queue. The algorithm keeps iterating until one of the partial paths reaches t.

We discover rules with a similar technique. For each example $(x, y) \in G$, we start the navigation from x. We keep a queue of not valid rules (Section 2.1), 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* [19], i.e., the estimated cost must be less than or equal to the actual cost. For example, for the shortest route problem, an admissible estimation is the straight-line distance to the goal for every node, as it is physically the smallest distance between any two points. 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 5.3. 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 other words, expanding r means adding a new atom to the body of r. 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 weight of any valid rule that is generated by expanding r. Given a rule and some expansions of it, we can derive the following.

Algorithm 1: RuDiK Rule Discovery.

input : G – generation set
input : V – validation set
input : $maxPathLen$ – maximum rule body length
output: R_{opt} – greedy set cover solution

$R_{opt} \leftarrow \emptyset$;
$N_f \leftarrow \{x | (x, y) \in G\}$;
$Q_r \leftarrow \texttt{expandFrontiers}(N_f)$;
$r \leftarrow \underset{r \in Q_r}{\text{argmin}}(w_m^*(r))$;
repeat
  $Q_r \leftarrow Q_r \backslash \{r\}$;
  if `isValid`(r) **then**
    $R_{opt} \leftarrow R_{opt} \cup \{r\}$;
  else
    `// rules expansion`
    if `length`$(r_{body}) < maxPathLen$ **then**
      $N_f \leftarrow \texttt{frontiers}(r)$;
      $Q_r \leftarrow Q_r \cup \texttt{expandFrontiers}(N_f)$;
  $r \leftarrow \underset{r \in Q_r}{\text{argmin}}(w_m^*(r))$;
until $Q_r = \emptyset \vee C_{R_{opt}}(G) = G \vee w_m^*(r) \geqslant 0$;
if $C_{R_{opt}}(G) \neq G$ **then**
  $R_{opt} \leftarrow R_{opt} \cup \texttt{singleInstanceRule}(G \backslash C_{R_{opt}}(G))$;
return R_{opt}

Lemma 5.4. *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 [3].

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 therefore estimate the coverage over the validation set to be 0 for any rule that can be further expanded, since expanding the rule may bring the coverage to 0.

Definition 5.5. 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 instead is equal to the actual marginal weight defined in Equation 5.1. Valid rules are not considered for expansion, therefore we do not need to estimate their weights since we know the actual ones. Given Lemma 5.4, we can easily see that $w_m^*(r) \leqslant w_m^*(r')$, for any r' expansion of r. Thus our marginal weight estimation is admissible.

We use the concept of *frontier nodes* for a rule r, namely $N_f(r)$. Given a rule r, $N_f(r)$ contains the last visited nodes in the paths that correspond to r_{body} from every example graphs

covered by r. As an example, given r_{body} = `child`(x, v_0), $N_f(r)$ contains all the entities v_0 that are children of x, for every $(x, y) \in G$. Expanding a rule r implies navigating a single edge from any frontier node. Algorithm 1 shows the modified set cover version that includes A^*-like rule generation. 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 from the KB all nodes (along with edges) at distance 1 from frontier nodes and returns the set of all rules generated by this one hop expansion. Such expansions are computed with single-hop SPARQL queries. 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, r is added to the output and it is not expanded (Line 8). If r is not valid, r 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 previously described greedy set-cover algorithm.

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). If the initial part of a path is not promising according to its estimated weight, the rest of the path will never be materialized. 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 leads to pruning unpromising rules. If a rule does not cover new elements in G and does not unbounded cover new elements in V, then its estimated marginal weight is 0 and will be pruned.

6 EXPERIMENTS

We implemented the above techniques in RuDiK, our system for Rule Discovery in Knowledge Bases. We carried out an extensive experimental evaluation of our approach and grouped the results in four main sub-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 in Machine Learning algorithms with representative training data; *(iv)* testing the role of the parameters in the system.

Settings. We ran experiments over the latest core facts derived from several KBs. All 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.

Parameters α and β of Equation 1 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 analyze the role of these parameters in Section 6.3.

Evaluation Metrics. We evaluated the effectiveness of RuDiK in discovering both positive and negative rules. The set of predicates for evaluation were chosen separately in the case of positive and negative rules from each KB. 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 some meaningful rules existed or not.

The evaluation of the discovered rules has been done according to the state of the art for rule quality evaluation [16]. 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.

Full test results, including KBs, induced rules, annotated examples and rules are available at http://bit.ly/2csROsc.

Table 1: Dataset characteristics.

KB	*Version*	*Size*	*#Triples*	*#Predicates*
DBPedia	3.7	10.056GB	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

6.1 Quality of Rule Discovery in RuDiK

The first set of experiments aimed at evaluating the accuracy of discovered rules over three popular KBs: DBPedia [6], Yago [26], and Wikidata [28]. Table 1 shows the characteristics of these KBs.

The size of the KB is important as loading the entire KB in memory is not feasible unless we either have HW with large amount of memory [8, 15], or we shrink the KB by eliminating all the literals [16]. In our approach, only a small portion of the KB is loaded in memory, thus we can afford to (i) discover rules with commodity HW resources, (ii) retain the literals, which are crucial for the literals comparison in our rules. While RuDiK mines rules for a given target predicate at a time, it could also discover rules over the entire KB given its set of predicates. This is further elaborated in Section 6.2.

Table 2: RuDiK Positive Rule Accuracy.

KB	*Avg. Run Time*	*Avg. Precision*	*# Annotations*
DBPedia	34min, 56sec	**63.99**%	139
Yago 3	59min, 25sec	**62.86**%	150
Wikidata	2h, 21min, 34sec	**73.33**%	180

Positive Rules RuDiK. We first evaluate the precision for the positive 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 as

explained above, and then average values over all induced rules. Table 2 reports precision values, along with predicates average running time. Column *# Annotations* reports the size of the sample of manually annotated triples.

Results show that the more accurate is the KB, the better is the quality of the induced rules. WIKIDATA contains very few errors, since it is manually curated, while DBPEDIA and YAGO are automatically generated by extracting information from the Web, hence their quality is significantly lower. Some predicates, such as `academicAdvisor`, `child`, and `spouse`, have a precision above 95% in all KBs, but average precision values are brought down by few predicates, such as `founder`, where meaningful positive rules probably do not exist at all. In other terms, when a rule existed, the system was able to find it, but it was not able to recognize cases where no positive rules existed (indeed it is not possible to derive a founder from the information on persons and companies).

The running time is influenced by the size of the KB with the number of triples and the number of predicates. The more predicates we have in the KB, the more alternative paths we test while traversing the graph. Another relevant aspect is the target predicate involved. Some entities have a huge 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 as we traverse a large number of edges. Parameter $maxPathLen$ also impacts the running time. The longer the rule, the bigger is the search space, as we discuss in Section 6.3.

Table 3: RuDiK Negative Rule Accuracy.

KB	*Avg. Run Time*	*# Pot. Errors*	*Precision*
DBPEDIA	19min, 40sec	499 (84)	**92.38**%
YAGO 3	10min, 40sec	2,237 (90)	**90.61**%
WIKIDATA	1h, 5min, 38sec	1,776 (105)	**73.99**%

Negative Rules RuDiK. We evaluated negative rules as the percentage of correct errors discovered for the top 5 predicates in each KB. Table 3 shows, for each KB, the total number of potential erroneous triples discovered with our output rules, whereas the precision is computed as the percentage of actual errors among potential errors. Numbers in brackets show the sample of unknown triples manually annotated.

Negative rules have better accuracy than positive ones. This is due to the fact that negative rules exist more often then positive rules. YAGO has the highest number of errors; for example, there are $9,057$ cases in the online YAGO where a child is born before the parent. 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. Temporal information are usually expressed through dates, years, or other primitive types that 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 is simply all the 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 SPARQL engines.

6.2 Comparative Evaluation

We compared our methods against AMIE [16], the state-of-the-art positive rule discovery system for KBs. AMIE loads the entire KB into memory and discovers positive rules for every predicate. It then outputs all possible rules that exceed a given threshold and ranks them according to a coverage function.

Table 4: AMIE Dataset characteristics.

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

Given its in-memory implementation, AMIE went out of memory for the KBs of Table 1 on our machine. Thus, we used the modified versions of YAGO and DBPEDIA from the AMIE paper [16], 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 4 reports the number of triples added to the original AMIE dataset.

Positive Rules Comparison. We first compared against AMIE on its natural setting: positive rule discovery. AMIE takes as input an entire KB, and discovers rules for every predicate in the KB. We adapted RuDiK to this setting 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.

For this evaluation, we plot the total cumulative number of new unique predictions (x-axis) versus the aggregated precision (y-axis) when incrementally including in the solution the rules according to their descending score. Figures 3 and 4 report the results on YAGO and DBPEDIA, respectively.

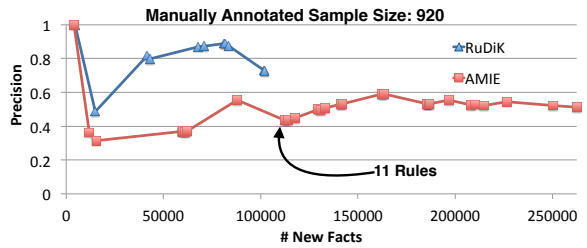

Figure 3: Accuracy for new facts identified by executing rules according to descending score on Yago 2 (no literals).

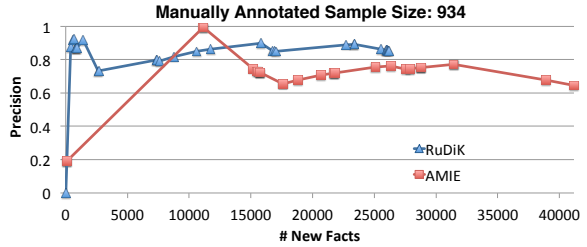

Figure 4: Accuracy for errors identified by executing rules according to descending score on DBPedia (no literals).

Rules from AMIE produce more predictions, but with a 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 top k rules. 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. AMIE is not designed to discover negative rules, and can mine rules only for predicates explicitly stated in the KB. To use it as a baseline, we created a set of negative examples as explained in Section 4.2 for each predicate in the top-5. To let AMIE mine this information, for each negative 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 run AMIE on these new predicates.

Table 5: 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%**

Table 5 shows that RuDiK outperforms AMIE in both cases with a precision gain of almost 20%. The drop in quality for RuDiK w.r.t. the ones showed in Section 6.1 is because we are using KBs without literals. Numbers in brackets show the number of triples manually annotated.

Running Time. Running times for AMIE are different from [16], where it was run on a 48GB RAM server. 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.

Table 6: 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 6 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 in the KB.

Other Systems. We found other available systems to discover rules in KBs. In [2], the system discovers rules that are less general than our approach; on YAGO 2, it discovers 2K new facts with a precision lower than 70%, while the best rule we discover on YAGO 2 already produces more than 4K facts with a 100% precision. A recent system [8] implements AMIE algorithm with a focus on scalability. They do not modify the mining part, but split the KB into multiple cluster nodes to run in parallel. The output is the same as AMIE. We did not compare with classic Inductive Logic Programming systems [11], as these are already significantly outperformed by AMIE both in accuracy and running time.

6.3 Internal Evaluation

We briefly outline the most relevant findings when studying the impact of individual components in RuDiK. Full results on the internal evaluation can be found in the technical report online at http://bit.ly/2bDZO9F.

LCWA. We study the effect of the LCWA assumption for the generation of negative examples. Given a predicate p, we

Table 7: Effect of Negative Examples Generation Strategy on DBPedia.

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

tested three different generation strategies: RuDiK strategy (as detailed in Section 4.2), 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 by a predicate different from p). Table 7 reports the results for the discovered rules. *Random* and *LCWA_Random* show similar behavior, with a slightly better precision than RuDiK. This is because whenever we randomly pick examples from the Cartesian product of subject and object, the likelihood of picking entities from a different time period 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 different rules. Rules such as $\texttt{parent}(a, b) \Rightarrow \texttt{notSpouse}(a, b)$ are not generated with random strategies, since the likelihood of picking two people that are in a parent relation is very low. While we use LCWA in our default configuration, the results show the robustness of the algorithm w.r.t. the quality of the negative examples.

Literals. Including literals in the mining is especially beneficial for negative rules, where we double the number of errors discovered with a 10% increase in precision.

Path Length. $maxPathLen = 3$ is the optimal value for the parameter: with smaller values we lose several meaningful rules, and with bigger values we do not gain in precision while the execution time increases by an order of magnitude.

Weight Parameters. We empirically validated the choice of the values used for α and β in positive and negative settings. High α leads to higher recall with lower precision. In both positive and negative settings, the variation in performance for $\alpha \in [0.1, 0.9]$ is anyway limited ($\leqslant 12\%$), showing the robustness of the set cover problem formulation.

Search. The A^* search algorithm and pruning reduces by more than 50% the average running time with peaks of an order of magnitude compared to a baseline algorithm that generates all possible rules and applies the greedy algorithm on it.

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. Correct rules oftentimes are not among the top-10 ranked, and we found cases where meaningful rules are below the 100^{th} position.

7 RELATED WORK

A significant body of work has addressed the problem of discovering constraints over relational data. Dependencies are discovered over the attributes of a given schema and encoded into formalisms, such as Functional Dependencies [3, 20] and Denial Constraints [9]. However, these techniques cannot be applied to KBs for three main reasons: *(i)* the schemaless nature of RDF data and the open world assumption; *(ii)* 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; *(iii)* even when the algorithms are designed to support more errors [1, 21], there are scalability issues on large RDF dataset: a direct application of relational database techniques on RDF KBs requires the materialization of all possible predicate combinations into relational tables. Recently, Fan et. al. [14] laid the theoretical foundations of Functional Dependencies on Graphs. However, their language covers only a portion of our negative rules and does not include smart literal comparisons, which we have shown to be useful when detecting errors in KBs.

RuDiK is the first approach that is generic enough to discover both positive and negative rules in RDF KBs. Rule mining approaches designed for positive rule discovery in RDF KBs, such as AMIE [16] and OP algorithm [8], load the entire KB into memory prior to the graph traversal step. This is a constraint for their applicability over large KBs, and neither of these two approaches can afford value comparison. In contrast to them, by generating the graph on-demand, RuDiK discovers rules on a small fraction of the KB. This makes it scalable and the low memory footprint enables a bigger search space with rules that can have literal comparisons. We showed in the experimental section how RuDiK outperforms AMIE both in final accuracy and running time. Finally, [2] recommends new facts by using association rule mining techniques. Their rules are made only of constants and are therefore less general than the rules generated by RuDiK.

ILP systems such as WARMR [11] 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 these two systems [16]. Sherlock [24] is an ILP system that extracts first-order Horn Rules from Web text. While extending RuDiK to free text is an interesting future work, the statistical significance estimate used by As in other ILP systems inspired from Association Rule Mining [4], also Sherlock relies on thresholds for support and confidence that are non-trivial to set (Section 6.2). We avoid thresholds in RuDiK and rely on a set cover problem formulation that outputs only rules contributing to the coverage of the generation set while minimizing the coverage of the validation set.

8 CONCLUSION

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

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

negative rules not only identify potential errors in KBs, but also generate representative training data for ML algorithms.

Interesting open questions are related to the interactive discovery of the rules. It is not clear if and how it is possible to drastically reduce the runtime of the discovery with sampling of the input training instances while not compromising on the quality of the mined rules. Another interesting future direction is to discover more expressive rules that can exploit temporal information through smarter analysis of literals [1]. For instance, "if two person have age difference greater than 100 years, then they cannot be married" is an example rule that requires non-trivial analysis of temporal information.

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] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, 1993.

[5] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction for the web. In *IJCAI*, pages 2670–2676, 2007.

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

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *NIPS*, pages 2672–2680, 2014.

[19] 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.

[20] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

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

[22] B. Min, R. Grishman, L. Wan, C. Wang, and D. Gondek. Distant supervision for relation extraction with an incomplete knowledge base. In *HLT-NAACL*, pages 777–782, 2013.

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

[24] S. Schoenmackers, O. Etzioni, D. S. Weld, and J. Davis. Learning first-order horn clauses from web text. In *Empirical Methods in Natural Language Processing*, pages 1088–1098, 2010.

[25] 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.

[26] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A core of semantic knowledge unifying wordnet and wikipedia. In *WWW*, pages 697–706, 2007.

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

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