Discovery of Positive and Negative Rules in Knowledge-Bases

Stefano Ortona*
University of Oxford
stefano.ortona@cs.ox.ac.uk

Vamsi Meduri
Arizona State University
vmeduri@asu.edu

Paolo Papotti
Arizona State University
ppapotti@asu.edu

ABSTRACT

We present RuDiK, a system for the discovery of declarative rules over knowledge-bases (KBs). RuDiK output is not limited to rules that rely on "positive" relationships between entities, such as "if two persons have the same parent, they are siblings", as in traditional constraint mining for KBs. On the contrary, it discovers also negative rules, i.e., patterns that lead to contradictions in the data, such as "if two person are married, one cannot be the child of the other". While the former class is fundamental to infer new relationships in the KB, the latter class is crucial for other tasks, such as error detection in data cleaning, or the creation of negative examples to bootstrap learning algorithms. The algorithm to discover positive and negative rules is designed with three main requirements: *(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 outperform previous proposals in terms of efficiency and that it discovers more effective rules for the application at hand.

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 [29], DeepDive [26], Yago [27], 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, oftentimes from the Web, with minimal or no human intervention. This leads to two main problems. First, errors are propagated from the sources, or introduced by the extractors, leading to false facts in the KB. Second, usually KBs do not limit the information of interest with a schema that clearly defines instance data. The set of predicates is unknown a-priori, and adding facts defined on new predicates is just a matter of inserting new triples in the KB without any integrity check. Since *closed world assumption* (CWA) does not hold [13, 17], we cannot assume that a missing fact is false, but rather we should label it as *unknown* (*open world assumption*).

As a direct consequence, the amount of errors and incompleteness in KBs is significantly larger than in classic databases [28]. 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 KBs to improve the quality of the data [2, 8, 17]. However, these approaches so far have focused on the discovery of rules to derive new facts only, while for the first time we target the discovery of two different types of rules: *(i) positive rules*, used to enrich the KB with new facts and thus increase its coverage of the reality; *(ii) negative rules*, used to spot logical inconsistencies and identify erroneous triples.

Example 1: Consider a KB with information about parent and child relationships. A positive rule r_1 can be:

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

which 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:

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

The above rule states that a person b cannot be child of a if a was born after b. By instantiating the above rule for the `child` facts in the KB, we may discover 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 in order to be enforced, and there are thousands of rules in a large KB with hundreds of predicates [18]. Other than enriching and cleaning KBs, negative rules support other use cases. We will show how negative rules can improve Machine Learning tasks by providing meaningful training examples [24, 26].

*Work done while at Arizona State University

Proceedings of the VLDB Endowment, Vol. 9, No. 12

However, three main challenges arise when discovering rules for KBs.

Errors. Traditional database techniques for rule discovery make the assumption that data is either clean or has a negligible amount of errors [3, 9, 20, 30]. We will show that KBs present a high percentage of errors and we need techniques that are noise tolerant.

Open World Assumption. KBs contain only positive statements, and, without CWA, we cannot derive negative statements as counter examples – classic databases approaches rely on the presence of positive and negative examples [11, 23] or a fixed schema given as input.

Volume. Existing approaches for discovery of positive rules in KBs assume that data can fit into memory which is not realistic given the large and increasing size of the KB [2, 17]. More recent approaches also rely on in-memory algorithms and try to solve this problem with distributed architectures [8, 16].

We advocate that a rule discovery system should be designed to discover *approximate rules* since errors and incompleteness are in the nature of the KBs. Also, the in-memory constraint not only reduces the applicability of the system, but also forces limitations on the *expressive power* of the language in order to reduce the search space. In fact, the larger is the number of patterns that can be expressed in the rules, the larger is the number of new facts and errors that can be identified, and with increasing precision. However, this comes with a computational cost, as the search space quickly becomes much larger. For this reason, existing approaches for mining positive rules prune aggressively the search space, and rely on a simple language [2, 8, 17].

We present RuDiK (Rule Discovery in Knowledge Bases), a novel system for the discovery of positive and negative rules over KBs that addresses the challenges above. RuDiK is the first system capable of discovering both approximate positive and negative rules without assuming that the KB fits into memory by exploiting the following contributions.

Problem Definition. We formally define the problem of rule discovery over erroneous and incomplete KBs. The input of the problem consists of a target predicate, from which we identify sets of positive and negative examples. In contrast to the traditional ranking of rules based on a measure of support [11, 17, 25], our problem definition aims at the identification of a 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. Moreover, to effectively steer the algorithm towards useful rules, the input examples must have properties, such as the existence of at least a predicate between pairs of entities in each example. 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 the rules in the algorithm by judiciously using the memory. The algorithm incrementally materializes the KB as a graph, and discovers rules by navigating its paths. To reduce the use of resources, at each iteration we follow the most promising path in the A^* traversal fashion, allowing the pruning of unpromising paths. By materializing only the portion of the KB that is needed for the discovery, and generating nodes and edges *on demand*, the disk is accessed only whenever the navigation of a specific path is required (Section 5). The greedy traversal of the search space reduces the running time by an order of magnitude in the best case, and its low memory footprint enables rule discovery for KBs that do not fit into memory.

We experimentally test the performance of rule discovery in RuDiK by using real-world datasets (Section 6). The evaluation is carried on three popular and widely used KBs, namely DBPedia [6], Yago [27], and Wikidata [29]. We show that our problem formulation together with our search algorithm 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. We also demonstrate how negative rules provide training examples for Machine Learning algorithms of quality comparable to examples manually crafted by humans.

2. PRELIMINARIES AND DEFINITIONS

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* is connected to an *object* via a *predicate*. 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 do not have a schema that defines allowed instance data. The set of predicates is unknown a-priori, and new predicates are added by inserting new 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 in order to define classes, domain/co-domain types for predicates, and relationships among classes to check integrity and consistency, in most KBs – including the ones we use 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 first-order logical formulas in KBs. More specifically, we target the discovery of *Horn Rules*. 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 simply *positive rule*, as it generates new positive facts, while the latter *goal clause* or *negative rule*, as it identifies false statements. 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 rel is a predicate, and a and b are either variables or entities. Given a Horn Rule r, we define r_{body} and r_{head} as the body and the head of the rule, respectively. We define the variables appearing in the head of the rule as the *target variables*. For the sake of presentation, we will also write negative rules as definite clauses by rewriting a body atom in its negated form in the head. The result is a logically equivalent formula that emphasizes the generation of negative facts.

Example 2: Rule r_1 in Example 1 is a traditional positive rule, where new positive facts are identified with target variables a and b. Rule r_2 is a negative rule to identify errors, as in denial constraints for relational data [9]. However, for other applications, we can rewrite it as a definite clause to derive false facts from the KB and obtain r_2':

$$\texttt{birthDate}(a, v_0) \wedge \texttt{birthDate}(b, v_i) \wedge v_0 > v_i \Rightarrow \neg\texttt{child}(a, b)$$

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 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 a KB kb and 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 such that if we substitute variables with entities in A, A holds over kb. Transitively, given a body of a rule r_{body} and a KB kb, we say that r_{body} can be instantiated over kb if every atom in r_{body} can be instantiated.

Following the biases introduced by other approaches for rule discovery in KBs [8, 17], we adopt also two constraints on the variables in the rules. 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. The connectivity constraint requires that every atom in a rule must be *transitively* reached 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, the repetition constraint limited to the body of a rule requires that each variable that is not a target variable must be involved in a join or in a literal comparison.

Language restrictions limit the algorithm output to a subset of plausible rules. We will show in Section 4 how these restrictions enable us to speed up the discovery process.

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 Horn 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 Horn 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}, the target variables are not bounded. 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 3: Given rule r_2' of Example 2 and a KB kb, 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 birthDate 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 birthDate 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 birthDate information 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 birthDate information 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 of paramount importance.

We can now define the coverage and the unbounded coverage for 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. RULE DISCOVERY

Our problem tackles the discovery of rules for a *target predicate* given as input. We characterize a predicate with two different 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. For example, in 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 that *are not* in a child relation. In case we want to identify errors in the child predicate, the sets of examples are the same, but they switch role. For the discovery of negative rules for child, G contains pairs of people not in a child relation and V contains examples of real pairs. We will explain in Section 4.2 how to generate these two sets for a given predicate. Note that our approach is not less generic than those for mining rules for an entire KB (e.g., [2, 17]): we can apply our setting for every predicate in the KB and compute rules for each of them (see Section 6.2).

We can now formalize the *exact discovery problem*.

Definition 1: Given a KB kb, two sets of pairs of entities

G and V from kb such that $G \cap V = \varnothing$, and a universe of Horn Rules R, a solution for the *exact discovery problem* is a subset R' of R such that:

$$R_{opt} = \underset{|R'|}{\operatorname{argmin}}(R' | (C_{R'}(G) = G) \wedge (C_{R'}(V) \cap V = \varnothing))$$

The exact solution is a set of rules that covers all examples in G, and none of the examples in V.

Example 4: Consider the discovery of positive rules for the predicate `couple` between two persons using as examples the Obama family. A positive example is (Michelle, Barack) and a negative example their daughters (Malia, Natasha). Given two positive rules:

$$\texttt{livesIn}(a, v_0) \wedge \texttt{livesIn}(b, v_0) \Rightarrow \texttt{couple}(a, b)$$
$$\texttt{hasChild}(a, v_i) \wedge \texttt{hasChild}(b, v_i) \Rightarrow \texttt{couple}(a, b)$$

The first rule states that two persons are a couple if the live in the same place, while the second states that they are a couple if they have a child in common. Both rules cover the positive example, but only the second rule does not cover the negative one as all of them live in the same place.

In the problem definition, the solution minimizes the number of rules in the output to avoid precise rules covering only one pair, which are not useful when applied on the entire KB. Note in fact that given a pair of entities (x, y), there is always a Horn Rule whose body covers only (x, y) by assigning target variables to x and y (e.g., $\texttt{hasChild}(Michelle, v_i) \wedge \texttt{hasChild}(Barack, v_i) \Rightarrow \texttt{couple}(Michelle, Barack)$).

Unfortunately this definition leads to poor rules because of the data problems in KBs. Even if a valid, general rule exists semantically, missing information 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. The exact solution may therefore be a set of rules where every rule covers only one example in G and none in V, ultimately leading to a set of overfitting rules.

3.1 Weight Function

Given errors and missing information in both G and V, we drop the requirement of perfectly 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, as we assume that errors are always a minority in the data. We model these ideas in a *weight* associated with a rule.

Definition 2: Given a KB kb, two sets of pair of entities G and V from kb such that $G \cap V = \varnothing$, 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 elements of G and none of V would have a weight of 0. The weight is made of two components normalized by the two parameters α and β. *1)* The first component captures the coverage over the generation set G – the ratio between the coverage of r over G and G itself. *2)* The second component aims at quantifying potential errors of r by using the coverage over V. The coverage over V is not divided by total elements in V because some elements in V might not have predicates stated in r_{body}. Thus we divide the coverage over V by the unbounded coverage of r over V.

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 a high α champions the recall as the discovered rules identify as many examples as possible.

Example 5: Consider again the negative rule r'_2 of Example 2 and two sets of pairs of entities G and V from a KB kb. The two components of w_r are computed as follow: *1)* the first component is computed as 1 minus the number of pairs (x, y) in G where x is born after y divided by the total number of elements in G; *2)* 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 date of birth (for both x and y) is available in kb.

Definition 3: 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 4: Given a KB kb, two sets of pair of entities G and V from kb where $G \cap V = \varnothing$, a universe of rules R, and a w weight function for R, a solution for the *approximate discovery problem* is a subset R' of R such that:

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

We can map this problem to the well-known weighted set cover problem, which is proven to be NP-complete [10]. The universe corresponds to G and the input sets are all the possible rules defined in R.

Since we want to minimize the total weight of the output rules, the approximate version of the discovery problem aims to cover all elements in G, and as few as possible elements in V. Since for each element $g \in G$ there always exists a rule that covers exactly only g (single-instance rule), an admissible solution is always guaranteed to exist. We expect the output to be made of some rules that cover multiple examples in G, with the remaining examples in G to be covered by single-instance rules.

4. RULE AND EXAMPLE GENERATION

This and the following Section are the main technical contributions of the paper. In this Section we describe how to generate the universe of all possible rules, while in the next Section we propose a greedy approximated solution to our weighted set cover problem.

In this Section we assume that examples are given, and then show how positive and negative examples can be computed. However, our approach is independent on how G and V are generated: they could be manually crafted by domain experts, with significant additional manual effort.

In the following we focus on the discovery of positive rules, i.e., having true facts in G and false facts in V. However, in

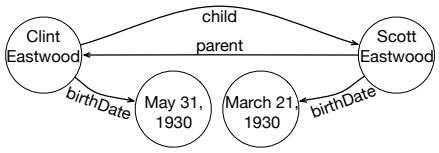

Figure 1: Graph Portion of DBPedia

the dual problem of negative rules discovery our approach remains unchanged, we just switch the role of G and V. The generation set becomes V (negative examples), while the validation set becomes G (positive examples).

4.1 Rule Generation

The first task we address is the generation of the universe of all possible rules R. The number of possible rules increases exponentially with the size of the KB and enumerating all of them leads to exploring a huge search space. While loading the KB in memory alleviates this problem [8, 17], modern KBs can easily exceed the size of hundreds of GBs, making a memory-based solution infeasible without significant hardware, such as a distributed platform [8, 16]. On the contrary, our rule generation technique inspects only a portion of the KB without losing any rule that is good w.r.t. our weighting scheme. Also, since we load only a small fraction of the database into main memory, we can afford a more expressive language compared to previous works.

Each rule in R must cover one or more examples in the generation set G. Thus the universe of all possible rules is generated by inspecting elements of G only. The smaller the size of G is, the smaller the search space for rule generation.

We translate a KB kb into a directed graph: entities and literals are nodes of the graph, while there is a direct edge from node a to node b for each triple $\langle a, rel, b\rangle \in kb$. Edges are labelled, where the label is the relation rel that connects subject to object in the triple. Figure 1 shows a portion of DBPedia [6] built from four different 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 *Scott Eastwood* → *Clint Eastwood* → *Scott 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 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 covered.

In other words, 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 of navigating 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, navigating an edge from a to b produces the atom $\texttt{rel}(a, b)$, while b to a produces $\texttt{rel}(b, a)$.

One can notice that for two entities x and y, there might exist infinite valid paths starting from x, since every node can be traversed multiple times. Therefore we introduce the $maxPathLen$ parameter that determines the maximum number of edges in the path. When translating paths to Horn Rules, $maxPathLen$ determines the maximum number of atoms that we can have in the body of the rule. This parameter is essential to constraint the search space.

Create Paths. Given a pair of entities (x, y), we retrieve from the KB all nodes at distance smaller than $maxPathLen$ from x or y, along with their edges. The retrieve 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 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.

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}. Our technique also generates single-instance rules: rules that cover only one example $(x, y) \in G$ by instantiating target variables a and b in the rule with x and y. Once the universe of all possible rules has been generated (along with coverages over G), computing coverage and unbounded coverage over V is just a matter of executing two SPARQL queries against the KB for each rule in the universe.

Generating More Atom Types

Literals comparison. In Section 2.1 we defined our target language, which, other than predicate atoms, includes literals comparison. Their scope is to enrich the language with comparisons among literal values other than equalities, such as "greater than". To discover such kind of 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 key point to introduce literals 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. Since in a single example graph the number of literals is usually small, the creation of a quadratic number of edges w.r.t. the number of literals is affordable. 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. For example, consider the negative rule:

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

The rule states that if a person a is born in a country that is different from b, then a cannot be 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 and aiming at meaningful rules, we use the `rdf:type` triples associated to entities. We therefore add an artificial 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 negative 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:

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

To discover such rules, we introduce a refinement of the rule generation. For a given rule r, we promote a variable v in r to an entity e iff for every $(x, y) \in G$ covered by r, v is always 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. It consists in 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*.This implies that we cannot take two entities that are not related by a property and assume that they form a negative example.

To mitigate this problem, and generate negative examples that are likely to be correct (true false facts), we are after entities that are more likely to be complete in their information. For this task, we exploit the *Local-Closed World Assumption* (*LCWA*) [13, 17]. 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`). KBs contain many non-functional predicates, we therefore extend the definition of LCWA by considering the dual aspect: if a KB contains one or more subject values for a given object and predicate, then it contains all values.

Now that we have entities that are likely to be complete, we could 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) such that x has some children in the KB that are not y, or y is the child of someone that is not x. By considering the union of the two LCWA aspects we do not restrict predicates to be functional (such as in [17]).

The number of negative examples generated with the above technique is extremely large; for a target `child` predicate it is close to the cartesian product of all the people having a child with all the people having a parent. However, to apply the same approach for positive and negative rules discovery, we require G and V to be of comparable sizes. Instead of randomly selecting a subset, we introduce a further constraint that significantly reduces the size of V. Given 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 $rel' \neq rel$. This intuition exploits the LCWA for predicates rather than for entities. If a KB contains a relation between two entities x and y, then it contains all relations between x and y. This further restriction has two main advantages: (i) it makes the size of V of the same order of magnitude of G (see Section 6), and (ii) it guarantees the existence of a path between x and y, for every $(x, y) \in V$. Since V becomes the generation set in the negative rules discovery, V must be small in size and it must guarantee the existence of a path between pairs of entities in each example. Such a property was already guaranteed in G, since pairs are always connected at least by the target predicate.

Example 6: 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*.

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 an A^* search fashion.

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: 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. Due to the first negative part of the weight, $w_m(r) \in [-\alpha, +\beta]$. 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 are 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 $\texttt{log}(k)$ 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?

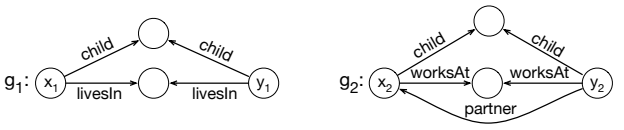

Figure 2: Two positive examples.

Example 7: 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. Differently from A^*, we do not stop when a rule (path) reaches the node y. 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*, 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 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 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.

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 unbound 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 its 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 instead is equal to the actual marginal weight defined in Equation 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 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 $r_{body} = \texttt{child}(x, v_0)$, $N_f(r)$ 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 algo-

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
1  R_opt ← ∅;
2  N_f ← {x|(x, y) ∈ G};
3  Q_r ← expandFrontiers(N_f);   // SPARQL
4  r ← argmin_{w*_m(r)}(r ∈ Q_r);
5  repeat
6      Q_r ← Q_r\{r};
7      if isValid(r) then
8          R_opt ← R_opt ∪ {r};
9      else
           // rules expansion
10         if length(r_body) < maxPathLen then
11             N_f ← frontiers(r);
12             Q_r ← Q_r ∪ expandFrontiers(N_f);   // SPARQL
13     r ← argmin_{w*_m(r)}(r ∈ Q_r);
14 until Q_r = ∅ ∨ C_{R_opt}(G) = G ∨ w*_m(r) ⩾ 0;
15 if C_{R_opt}(G) ≠ G then
16     R_opt ← R_opt ∪ singleInstanceRule(G\C_{R_opt}(G));
17 return R_opt
```

rithm maintains a queue of rules Q_r, from which it chooses at each iteration the rule with minimum approximated 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 away.

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 and some KB properties.

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.

In the following, 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 also set to 3 the $maxPathLen$ parameter, which indicates the maximum number of atoms admissible in the body of a rule. We analyze the role of these parameters in Section 6.4.

Evaluation Metrics. We evaluated the effectiveness of RuDiK in its ability to discover 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 as follows. For each 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 was done for both positive and negative rules according to the state of the art for rule quality evaluation [17]. If a rule was clearly semantically correct, we marked all its new predictions as true. If a rule was unknown, we randomly sampled 30 affected pairs, either as new facts (for positive rules) or as violations (for negative rules), and manually checked them. The *precision* of a rule is then computed as the ratio of true assertions out of true and false assertions.

The full suite of test results, together with all the KBs, induced rules, and annotated gold standard examples and rules is available online at `http://bit.ly/2csROsc`.

6.1 Quality of Rule Discovery

The first set of experiments aimed at evaluating the accuracy of discovered rules over three popular KBs: DBPedia [6], Yago [27], and Wikidata [29]. Table 1 shows the characteristics of the 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 [8,16], or we artificially shrink the KB by eliminating all the literals [17]. In our disk-based approach only a small portion of the KB is loaded in memory, therefore we discover rules with limited HW resources. Also, we can retain the literals, which are crucial for the value comparison in our rules. While RuDiK mines rules for a given target predicate, it discovers rules over the entire KB if the input is the universal set of predicates from the KB. This is further elaborated in Section 6.2.

Positive Rule Discovery. We first evaluate the precision for the positive rules with the top 5 predicates on the three KBs. 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

Table 1: Dataset characteristics.

KB	*Version*	*Size*	*#Triples*	*#Predicates*
DBPedia	3.7	10.056GB	68,364,605	1,424
Yago	3.0.2	7.82GB	88,360,244	74
Wikidata	20160229	12.32GB	272,129,814	4,108

Table 2: Positive Rule Accuracy.

KB	Avg. Run Time	Avg. Precision	# Manual
DBPEDIA	34min, 56sec	**63.99**%	139
YAGO	59min, 25sec	**62.86**%	150
WIKIDATA	2h, 21min, 34sec	**73.33**%	180

each rule as explained above, and then average values over all induced rules. Table 2 reports precision values, along with predicates average running time. The *Manual* column shows 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. Predicates like `academicAdvisor`, `child`, and `spouse` have a precision above 95% in all KBs, but average precision values are brought down by few predicates like `founder` where meaningful rules probably do not exist at all. In other terms, when a rule exists, the system was able to find, but it failed for cases where no positive rule seem to exist (indeed it is not possible to derive a founder from the information on persons and company in the KBs).

The running time is influenced by the size of the KB, both in terms of the number of triples and the number of predicates. RuDiK is slower in WIKIDATA which is the biggest KB. The more predicates we have in the KB, the more alternative paths we observe while traversing the graph. The second relevant aspect is the target predicate involved. Some entities have a huge number of outgoing and incoming edges, 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 need to traverse a large number of edges. The *maxPathLen* parameter also impacts the running time. The longer the rule, the bigger is the search space, as discussed in Section 6.4.

Negative Rule Discovery. 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 the 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 a negative rules exists more often then a positive rule. WIKIDATA shows lower numbers because it does not contain as many errors as DBPEDIA and YAGO: even though the discovered rules are almost correct, the percentage of actual errors identified is lower in WIKIDATA. YAGO is the KB with the highest number of errors. As an 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 vital role in discovering negative rules. In fact in many cases 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.

Table 3: Negative Rule Accuracy.

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

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 rule discovery method against AMIE [17], the state-of-the-art positive Horn Rule discovery system for KBs. AMIE loads the entire KB into memory and discovers positive rules for every predicate in the KB. It then outputs all possible rules that exceed a given threshold and ranks them according to a coverage function.

Given the in-memory implementation, AMIE cannot handle large KBs and went out of memory for the KBs of Table 1. Thus, we used the modified versions of YAGO and DBPEDIA from the AMIE paper [17], which are devoid of literals and `rdf:type` facts.

Table 4: AMIE Dataset characteristics.

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

Removing literals and `rdf:type` triples drastically reduce the size of the KB. Since our approach needs type information (both for the generation of G and V and for the discovery of entities inequality atoms), we run AMIE on its original dataset, while we run our algorithm on the same dataset plus `rdf:type` triples. The last column of Table 4 shows how many triples we added to the original AMIE dataset.

Positive Rules. We first compared RuDiK 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 our system to simulate AMIE as follows: we first list all the predicates in the KB, and for each predicate that connects a *subject* to an *object* we computed the most common type for both subject and object. The most common type is the most popular `rdf:type` that is not super class of any other most popular type. We then ran our approach sequentially on every predicate, setting $maxPathLen = 2$ (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 head predicate of the 30 best rules output of AMIE.

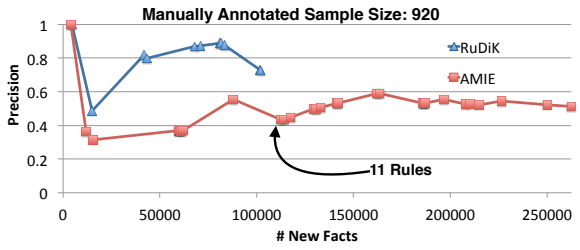

Figure 3: New Facts Accuracy on Yago

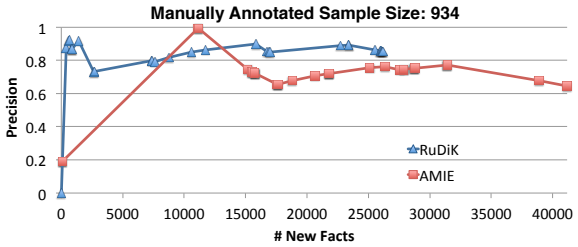

Figure 4: New Facts Accuracy on DBPedia

For this evaluation, we plot the total 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.

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. We used AMIE also to discover negative rules. AMIE is not designed to work in this setting, and can discover rules only for predicates explicitly stated in the KB. To use it as a baseline, we proceeded as follows. For each predicate in the top-5, we created a set of negative examples as explained in Section 4.2. To let AMIE mine this information, for each negative example we added a new fact to the KB 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	
KB	*# Errors*	*Precision*	*# Errors*	*Precision*
DBPEDIA	457 (157)	38.85%	148 (73)	**57.76%**
YAGO	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 the AMIE modified KBs which do not contain literals. Numbers in brackets show the sample of unknown triples manually annotated.

Running Time. We report the running time of AMIE compared to our approach. Numbers are different from [17], where AMIE was run on a 48GB RAM server. On our machine, AMIE could finish the computation only 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 reports the running time on different KBs. The first five KBs are AMIE modified versions, while YAGO 3* is

Table 6: Run Time vs AMIE.

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

complete YAGO, including 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 the KB, which easily fits in memory. However, when we deal with real KBs (YAGO 3*), the KB could not even be loaded due to out of memory errors. The last column reports the total time needed to compute `rdf:type` information for each predicate in the KB.

Other Systems. We found other available systems to discover rules in KBs. In [2], the system discovers rules that are less generic than our approach. On AMIE 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 Machine Learning Application

The main goal of this set of experiments is to demonstrate the applicability of our approach in providing Machine Learning (ML) algorithms meaningful training examples. We chose DeepDive [26], a ML framework to construct KBs. DeepDive extracts entities and relations from text articles via distant supervision. The key idea behind distant supervision is to use an external source of information (e.g., a KB) to provide training examples for a supervised algorithm. For example, DeepDive can extract mentions of married couples from text documents. In such a scenario DeepDive uses as a first step DBPEDIA to label some pairs of entities as *true* positive (those pairs of married couples that can be found in DBPEDIA). Unfortunately, KBs directly provide positive examples only. Hence in DeepDive the burden of creating negative examples is left to the user.

In this set of experiments, we created negative examples with the rules obtained on DBPEDIA with RuDiK. We then

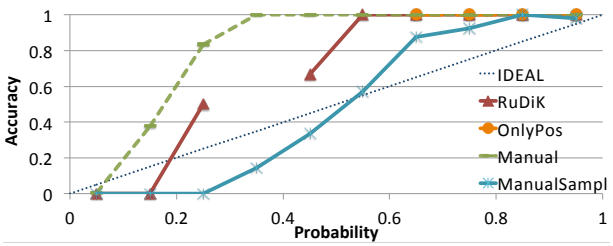

Figure 5: DeepDive Application 1K articles

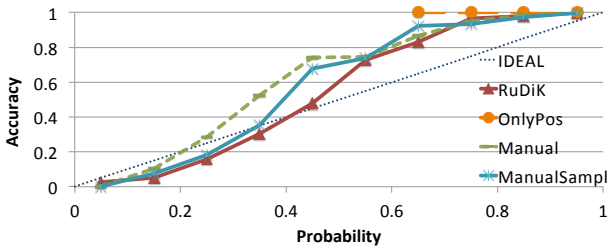

Figure 6: DeepDive Application 1M articles

compared the output of DeepDive upon its spouse example trained with different sets of negative examples.

Figure 5 shows DeepDive accuracy plot run on 1K input documents. The accuracy plot shows the fraction of correct positive predictions over total predictions (y-axis), for each output probability value (x-axis). The perfect algorithm, 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 is our evaluation metric. The figure shows 4 lines other than the ideal one. RuDiK is the output of DeepDive using our approach to generate negative examples. `OnlyPos` uses only positive examples from DBPEDIA, `Manual` uses positive examples from DBPEDIA and manually defined rules to generate negative examples, while `ManualSampl` uses only a sample of the manually generated negative examples in size equal to positive examples. We observe that `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 and labels everything as false. `ManualSampl` is the clear winner, while our approach suffers from the absence of data: over the input 1K articles, we could find only 20 positive and 15 negative examples from DBPEDIA. The lack of evidence in the training data also explains the missing points for RuDiK in the chart, where there were no predictions in the probability range 25-45%.

When we extend the input to 1M articles, things change drastically (Figure 6). All the three approaches except `OnlyPos` can successfully drive DeepDive in the training, with the examples provided with RuDiK leading to a slightly better result. This is because of the quality of the negative examples: our rules generate representative examples that help DeepDive in understanding 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 behaviour and provide negative examples at zero cost.

6.4 Internal Evaluation

In this last section we sketch the impact of individual components in RuDiK. Full results on the internal evaluation are not reported due to space reasons but can be found in the technical report online at `http://bit.ly/2bDZO9F`.

Here we briefly outline some of the most relevant findings. *1)* We show the benefits of including literals in the mining especially for negative rules, where we double the number of errors discovered with a 10% increase in precision. *2)* We show that 3 is the optimal value for the $maxPathLen$ parameter: with smaller values we lose several meaningful rules, and with bigger values we do not gain in precision. *3)* We empirically validate the choices of α and β for both positive and negative settings. *4)* We demonstrate the validity of our LCWA-based generation of negative examples, which clearly outperforms random generation strategies in terms of accuracy. *5)* We quantify the benefits of A^* search algorithm and pruning, where on an average the running time is halved with peak of an order of magnitude. *6)* We show the increase in precision of our set cover problem formulation w.r.t. a ranking based solution. Correct rules oftentimes are not among the top-10 ranked, and we show cases where meaningful rules are below the 100^{th} position.

7. RELATED WORK

Our work uses techniques from dependencies discovery in relational databases, KBs construction, and mining graph patterns. We review some of the most relevant works below.

A significant body of work has addressed the problem of discovering constraints over relational data. Due to the presence of a pre-defined schema, dependencies are discovered over the attributes and encoded into formalisms such as Functional Dependencies (FDs) [3, 20, 30], Conditional FDs [14] and Denial Constraints (DCs) [9]. However, techniques for FDs and DCs discovery cannot be applied to RDF databases for three main reasons: *(i)* the schema-less 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], scalability issues on large RDF dataset: a direct application of relational database techniques on RDF KBs requires the materilization of all possible predicate combinations into relational tables.

Recently, Fan et. al. [15] 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 literals comparisons, which we have shown to be useful when detecting errors in KBs.

To the best of our knowledge, RuDiK is the first approach that is generic enough to discover both positive and negative rules in RDF KBs. Rule mining approaches specifically designed for positive rule discovery in RDF KBs, such as AMIE [17] and OP algorithm [8], load the entire KB into memory prior to the graph traversal step. This is a strong constraints for their applicability over large KBs, and neither of these two approaches can afford smart literal comparison. In contrast to them, RuDiK is disk-based. By generating the graph on-demand, it can discover rules on a small fraction of the KB examples. This makes our approach 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. In contrast with recent approaches [16], our algorithm was designed to run on a single node in a commodity machine. We therefore have not tested our running time against a distributed environment. Finally, [2] recommends new facts to be added to the KB 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 closed world assumption and re-

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

quire the definition of positive and negative examples. AMIE outperforms these two systems [17], but shows scalability issues when dealing medium-size KBs. Moreover, classic ILP systems assume high-quality, error-free training examples as input. We showed how this assumption does not hold in KBs. Sherlock [25] is an ILP system that extracts first-order Horn Rules from Web text. While making RuDiK extensible to free text rather than relying on a well-defined KB is an interesting future work, the statistical significance estimate used by Sherlock needs a threshold to discover meaningful rules. Several ILP systems inspired from Association Rule Mining [4] also use thresholds for support and confidence that are non-trivial to set (Section 6.2). We avoid the use of a threshold in RuDiK and rely on a weighted 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

In this paper we presented RuDiK, a disk-based rule discovery system that mines both positive and negative declarative rules on RDF KBs. Positive rules identify new valid facts for the KB, while negative rules are useful to 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 negative rules not only identify potential errors in the KBs, but also discover new false facts that can serve as representative training data to ML algorithms.

Interesting open questions are related to the support of 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 people are not born in the same century, then they cannot be married" is an example rule that requires non-trivial analysis of temporal information.

9. 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, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *TODS*, 23(5):683–698, 2011.

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

[16] 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.

[17] 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.

[18] 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.

[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] M. Richardson and P. Domingos. Markov logic networks. *Machine learning*, 62(1-2):107–136, 2006.

[25] 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.

[26] 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.

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

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

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

[30] C. Wyss, C. Giannella, and E. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *DaWaK*, pages 101–110, 2001.