Affordable Discovery of Positive and Negative Rules in Knowledge-Bases

US

ABSTRACT

We present KRD, a system for the discovery of declarative rules over knowledge-bases (KBs). KRD does not limit its search space to rules that rely on "positive" relationships between entities, such as "if two persons have the same parent, they are siblings", as in traditional mining of constraints for KBs. On the contrary, it extends the search space to discover also negative rules, i.e., patters 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 error detection in data cleaning, or for the creation of negative examples when bootstrapping learning algorithms.

The main technical challenges addressed in this paper consist in enlarging the expressive power of the considered rules to include comparison among constants, including disequalities, and in designing a disk-based discovery algorithm, effectively dropping the assumption that the KB has to fit in memory to have acceptable performance. To guarantee that the entire search space is explored, we formalize the mining problem as an incremental graph exploration. Our novel search strategy is coupled with a number of optimization techniques to further prune the search space and efficiently maintain the graph. Finally, in contrast with traditional ranking of rules based on a measure of support, we propose a new approach inspired by set cover to identify the subset of useful rules to be exposed to the user. We have conducted extensive experiments using both real-world and synthetic datasets to show that KRD outperform previous proposals in terms of efficiency and that it discovers more effective rules for the application at hand.

1. INTRODUCTION

Example 1. *Given a KB that contains information about parent and child relationships, we can discover the following positive rule:*

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

which states that if a is parent of b, then b is child of a. On the other hand, a negative rule may be:

$$\texttt{birthDate}(a, v_0) \wedge \texttt{birthDate}(b, v_1) \wedge v_0 > v_1 \Rightarrow \neg\texttt{child}(a, b)$$

Such rule expresses the concept that b cannot be child of a if a was born after b.

2. PRELIMINARIES AND DEFINITIONS

Talk about KBs. Describe what entities and literals are

2.1 Language

Horn Rule. Given the definition of a *valid* rule (each variables appearing twice and each variable connected transitively to every other variable) and also give the definition of a valid body (a and b at least once, other variables twice). Extension of predicates with inequalities. Define instantiation of an atom.

A Horn Rule r has the form $A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow \texttt{r}(a, b)$, where $A_1 \wedge A_2 \wedge \cdots \wedge A_n = r_{body}$ is the *body* of the rule.

2.2 Coverage

Given a pair of entities (x, y) from the 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) iff r_{body} can be instantiated over the KB by substituting a with x and b with y. 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, $C_r(E) = \{(x, y) \in E | (x, y) \models r_{body}\}$.

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 substituting each atom in r that contains either variable a or b with a new atom where the other variable that is not a or b is substituted with another unique variable. As an example, given $r_{body} = \texttt{rel}(a, b)$, $r^*_{body} = \texttt{rel}(a, v_1) \wedge \texttt{rel}(v_2, b)$. Paolo: I suggest to have $rel_3(a, b)$ to avoid the confusion raised by cartesian product Stefano: Better now?Paolo: to me this is still unclear, the way that is written it applies for all combinations of a and b universal variables, no matter if they are related or not. Therefore it seems the cartesian product. The same seems to be stated in the following example. Adding a small instance in the intro will probably clarify things 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}, $U_r(E) = \{(x,y) \in E | (x,y) \models r^*_{body}\}$.

Example 2. *Given the negative rule r of Example 1 and a KB K, we denote by E the set of all possible pairs of entities in K. The coverage of r over E ($C_r(E)$) is the set of all pairs of entities (x,y) where both x and y have the* `birthDate` *information and x is born after y, while the unbounded coverage of r over E ($U_r(E)$) is the set of all pairs of entities (x,y) where both x and y have the* `birthDate` *information, no matter what the relation is between the two birth dates.*

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 from the KB for either x or y, we cannot say whether x was born before or after y, therefore we cannot be sure that the negative rule of Example 1 does not cover (x,y). Instead if both x and y have the `birthDate` information and x was born before y, we can affirm that the negative rule of Example 1 does not cover (x,y). Given that modern KBs are largely incomplete (REFERENCE), discriminating between missing and conflicting information becomes of paramount importance.

Similarly, the coverage and the unbounded coverage for a set of rules $R = \{r_1, r_2, \cdots, r_n\}$ is the union of individual coverages:

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

Our problem is the discovery of positive (negative) rules for an input given relation. We uniquely identify a relation with two different sets of pair of entities. G – *generation set.* G contains good examples for the relation that we are trying to discover (G contains examples of parents and children if we are discovering positive rules for a child relation). V – the validation set. V contains counter examples for the target relation (pairs of people that are not in a child relation). We will explain in Section 3.2 how to generate these two sets for a given relation. Note that our approach is not less generic than those for mining rules for the entire KB (e.g., [1, 7]): it is true that we require a target relation as input, however we can generically apply such setting for every relation in the KB and compute rules for each of them.

We can now formalize the *exact discovery problem.* Given a KB K, a set of pair of entities G, a set of pair of entities V, and a universe of 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 ideal solution is a set of rules that covers all examples in G, and none of the examples in V. Note that given a pair of entities (x,y), we can always generate a Horn Rule whose body covers only (x,y) by assigning variable a to x and variable b to y.

Unfortunately, since the solution is not allowed to cover any element in V, in the worst case the exact solution may be a set of rules s.t. each rule covers only one example in G, making such set of rules difficult to use.

2.3 Weight Function

In order to allow flexibility and errors in both G and V, we drop the strict requirement of not covering any element of V. However, since covering elements in V is an indication of potential errors, we want to limit the coverage over V to the minimum possible. We therefore define a *weight* to be associated with a rule.

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

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

with $\alpha, \beta, \gamma \in [0,1]$ and $\alpha + \beta + \gamma = 1$. The weight is a value between 0 and 1 that captures the *goodness* of a rule w.r.t. G and V: the better the rule, the lower the weight – perfect rule would have a weight of 0. The weight is made of three components normalized by the three parameters α, β, γ. *(i)* The first component captures the coverage over the generation set G – the ratio between the coverage of r over G and G itself. Note that if r covers all elements in G, then this component is 0 because of the subtraction from 1. *(ii)* The second component aims to quantify potential errors of r, or rather the coverage over V. The coverage over V is not divided by total elements in V, because for those elements in V that do not have relations stated in r we cannot be sure that such elements are not covered by r. Thus we divide the coverage over V by the unbounded coverage of r over V. Ideally this number is close to 0. *(iii)* The last element of the weight captures how many elements of V have the information stated by relations in r. The more elements in V are unbounded covered by r, the better we can judge the rule w.r.t. V. This element is close to 0 when r unbounded covers many elements of V. The parameters α, β, γ are used to give more relevance to some components. We would set a high β if we want to discover rules with high precision that identifies few mistakes, or we would set a high α if we are more interested in recall and the discovered rules should identify as many examples as possible.

Example 3. *W.r.t. the negative rule r of Example 1, given two sets of pair of entities G and V the three components of w_r are computed as follow: (i) the first component is computed as 1 minus number of pairs (x,y) in G where x is born after y divided by the number of elements in G; (ii) 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 the KB; (iii) the last component is computed as 1 minus number of pairs (x,y) in V where the date of birth (for both x and y) is available in the KB divided by the total number of elements in V.* Paolo: I think this would be more useful by referring to data in an example with micro KB

Similarly, the weight for a set of rules R is defined as:

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

Assigning a weight to one or multiple rules allows us to take into consideration an important aspect of modern KBs: the presence of errors. We will show in the experimental evaluation that very rarely rules have a 0 coverage over the validation set, and very often good rules have a significant coverage over V. The exact discovery problem implies the

absence of errors in the input KB, unfortunately such assumption is too strong for modern KBs that are automatically built(REFERENCE).

2.4 Problem Definition

We can now state the approximate version of the problem.

Given a KB K, two sets of pair of entities G and V from K where $G \cap V = \emptyset$, 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 a NP-Complete problem [4], where the universe is G and the 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 (x, y) in G there always exists a rule that covers only (x, y) (single-instance rule), an optimal output is always guaranteed to exist. We expect such output to be made of some rules that covers more than one example in G, and the remaining examples in G to be covered by single-instance rules. In the best-case scenario a single rule covers all elements in G and none of the elements in V.

Section 4 will describe a greedy polynomial algorithm to find a good solution for our problem. Paolo: Perhaps we should clarify the role of the queries in the bodies vs the role of the sets, and the fact that we do not have all the queries materialized because of their large number

3. RULES GENERATION

The first task we need to address is the generation of the universe of all possible rules R. Each rule in R must cover one or more examples of the generation set G. Recent KB rule mining approaches solve this aspect by loading the entire KB into main memory, and then discovering connections between entities by aggressively indexing KB triples on subject, object, and predicate [7](ANOTHER CITATION).Paolo: It is not clear why these low level aspects are crucial here. We should have a more methodological –high level- comment if we want to explain the different approach, I think These approaches work well with relatively small KBs. Unfortunately, modern KBs can easily exceed the size of hundred Gigas, which makes them too big to be entirely loaded into memory unless the whole process is run on a very resourceful machine. We propose an alternative disk-based solution that loads into memory only portions of the KB that are needed for the target relation. Paolo: why don't use their algorithm on a single relation? this is only one aspect! even with one relation their algorithm will load all the subgraph induced at distance k from the relation of interest: still to huge? perhaps this comparison should be done later on?

Given the generation set G, we can discover the universe of rules by just inspecting G. In fact, a rule that does not cover any element of G will never be part of the optimal solution, since it does not give any contribution to the set cover problem.

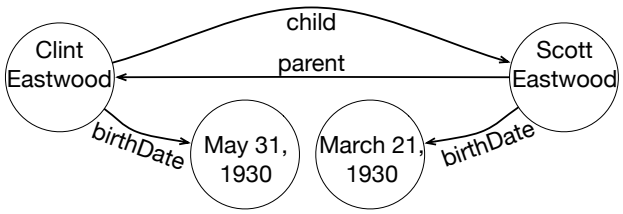

Figure 1: Graph Portion of DBPedia

Paolo: I think the first part should state that the generation problem is simple but has scalability issues. The number of possible rules is huge and we don't want to enumerate them. Creating rules is expensive, previous approach loaded it in memory. We now introduce our data structures and algorithms to face these two challenges while extending the language to be more expressive than previous work...

A KB K can be straightforwardly translated 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 $\langle a, rel, b \rangle \in K$. 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 that connects two person in a child and parent relationship, along with their dates of birth. The graph represents information of four KB triples (two birth dates, one parent and one child).

The body of a Horn Rule can be seen as a path in the graph. W.r.t. Figure 1, the body $\texttt{child}(a, b) \wedge \texttt{parent}(b, a)$ corresponds to the path *Clint Eastwood* $\rightarrow$ *Scott Eastwood* $\rightarrow$ *Clint Eastwood*. In Section 2.1 we defined valid rules. A valid body of a rule contains variables a and b at least once, and every other variable at least twice. In a valid body also each variable is connected transitively to every other variable. If we allow navigation of edges in any direction (no matter what the direction of the edge is on the graph), we can translate bodies of valid rules to paths on the graph. Given a pair of entities (x, y), a valid body corresponds to a path p on the graph that meets the following criteria:

1. p starts at the node x;
2. p touches y at least once;
3. p ends in x, in y, or in a different node that has been touched before.

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}. Therefore, given a pair of entities (x, y), we can generate bodies of all possible valid rules by simply computing all valid paths from x with a standard BFS. Note how the transitively connection between variables is always guaranteed by the construction properties of a path: for each node n in the path there exists a subpath that connects n to every other node in the path. The key is the ability of navigating each edge in any direction, which basically means turning the original directed graph into an undirected one.

Despite the navigation over an undirected graph, we still need to keep track of the original direction of the edges. This is essential when we want to translate a path to the body of a Horn Rule. In fact, if we have a direct edge rel from a to b, navigating the edge from a to b produces the atom $\texttt{rel}(a, b)$, while navigating the edge from b to a produces the

atom $\texttt{rel}(b, a)$. These two atoms are different, where the position of variables is determined by the original direction of the edge.

One should notice that for two entities x and y, there might exists infinite valid paths starting from x. Thus we introduce the $maxPathLen$ parameter that determines the maximum number of edges on 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 necessary to avoid the discovery of rules with infinite body length.

As said above, our rules generation approach does not need to load the entire KB in memory. We can load in memory just the portion of the graph that is needed. Paolo: not clear why other approaches cannot do simply the same, see above comment Given a pair of entities (x, y), we retrieve from the entire KB graph all the nodes at distance $maxPathLen - 1$ or less from x and y, along with their edges. Retrieving such nodes and edges can be done recursively: we maintain a queue of entities, and for each entity in the queue we fire a SPARQL query against the KB to retrieve all entities at distance 1 from the current entity (single hop queries). We add the new found entities to the queue iff they are distance less than $maxPathLen - 1$ from either x or y. The queue is initialised with x and y. By doing so we load into memory only a small portion of the KB, the only one needed to discover rules that cover (x, y). We will show in the experimental section that SPARQL engines are very fast at executing single hop queries.

The generation of the universe of all possible rules for a set G is then straightforward: for each element $(x, y) \in G$, we construct the portion of the graph as described above and compute all valid paths starting from x. By computing paths for every example in G, we can also compute the coverage over G for each rule. The coverage of a rule r is the number of elements in G where there exists a path that is equivalent to the body r. Path discovery will also generate single-instance rules: rules that cover only one example from G by instantiating variables a and b in the rule. Once the universe of all possible rules has been generated (along with coverages over G), we can compute coverages and unbounded coverages over V by simply executing two SPARQL queries against the KB for each rule in the universe. We will show in Section 4 how some queries can be avoided, as well as the generation of all possible rules.

3.1 Literals and Constants

Our language aims to be powerful enough to include smarter comparison among literal values other than equalities, such as greater than or less than. In the path discovery approach, this translates on having edges that connect literal values with such kind of comparisons. 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 creating comparisons for all literals in the KB is unfeasible. Since we discover paths for a pair of entities in isolation, thus the size of a graph for a pair of entities is relatively small, we can afford to compare all literal values within a single example graph. This implies the creation of a quadratic number of edges w.r.t. the number of literals in the graph. We will show in the experimental section that within a single example graph the number of literals is usually relatively small, thus the quadratic comparison affordable. Modern KBs include three types of literals: numbers, dates, and strings. Besides equality comparisons, we add '>','⩾','<','⩽' relationships between numbers and dates, and $\neq$ between all literals. These new relationships are treated as atoms: $x \geqslant y$ is equivalent to $\texttt{rel}(x, y)$, where $\texttt{rel}$ is equal to $\geqslant$.

We noticed that $\neq$ relation could be useful for entities as well other than literals. Think about the following 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 Paolo: this is a great example, but do we discover this rule in the exp?. The rule holds for most of the countries in the world. To consider inequalities among entities, we could add artificial edges among all pairs of entities in the graph. This strategy however, despite being inefficient for the high number of edges to add, would lead to many meaningless rules. We noticed that it is reasonable to compare two entities only when they are of the same type. All modern KBs include types information for every entity (often through the $\texttt{rdf:type}$ statement), therefore we use this information. We add an artificial inequality edge in the graph only between those pairs of entities of the same type. In the above rule it makes sense to compare x and b because they are both countries.

As a last extension of our language, we also discover rules with constants. For a given rule r, we promote a variable v in r to a constant c iff for every $(x, y) \in G$ covered by r, v is always instantiated with the same value c. Suppose that for the above negative rule for president, all examples in G are people connected to the country *U.S.A.*. We can then promote variable b to constant *U.S.A.*, generating the rule:

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

3.2 Input Examples Generation

A crucial point for our approach is how to generate the two input sets G and V, in order to generate and validate rules.

For an input KB K and a relation $rel \in K$, we automatically generate a set of *positive* examples (P) and a set of *negative* examples (N). Paolo: i would use generation vs validation to be general and consistent with first part Positive examples represent good examples for the target relation rel. They are easy to generate: they consist of all pairs of entities (x, y) such that $\langle x, rel, y\rangle \in K$ (all pairs of entities in a child relation if rel=child). Negative examples represent counter examples for the target relation and they are slightly more complicated to generate, since the closed world assumption does not longer hold in a KB. Differently from classic database scenarios, we cannot assume that what is not stated in a KB is false. Because of incompleteness, everything that is not stated in a KB is *unknown* rather false. In order to generate negative examples we make use of a popular technique for KBs: *Local-Closed World Assumption* (*LCWA*) [6, 7]. LCWA states that if a KB contains one or more object values for a certain subject and relation, then it contains all possible values (if a KB contains one or more children of Clint Eastwood, then it contains all the children). This is definitely true for *functional* relations (relations such as *capital* where the subject can have at most one object value), while it might not hold for non-functions

(*child*). KBs contain many non-functions relation, we therefore extend the definition of LCWA by considering the dual aspect: if a KB contains one or more subject values for a certain object and relation, then it contains all possible values.

To generate negative examples we then take the union of the two LCWA aspects: for a given relation 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 $x \neq x'$. 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 taking the union we do not restrict relations to be functions (such as in [7]).

With this technique however, the number of negative examples could be very large (in the child relation is nearly the cartesian product of all the people having either a child or a parent). We want to restrict the size and make it comparable to the size of positive examples. Paolo: argh! why? my intuition is that we want to exploit the semantics of the KB to get REAL/realistic rules. These are crucial to identify errors and create correct negative examples We therefore introduce another constraint, that significantly shrinks the size of negative examples: we require x and y to be connected by a relation in the original KB. This intuition comes from how modern KBs are built: there is an automatic process that extracts data from some input sources according to pre-defined rules. If x and y are in a relation in the KB, then the KB construction should have been able to generate all possible relations between x and y. If x and y are in one or more relations that is not the target relation, then most likely x and y are not in the target relation. This further restriction has multiple advantages: on the one hand it makes the size of negative examples of the same order of magnitude of positive examples (see Section 5), on the other hand it guarantees the existence of a path between x and y, for every (x, y) in the negative examples set. For positive examples, the existence of a path was already guaranteed since pairs in the positive examples set are connected by the target relation.Paolo: why this existing relation is important?

Once we generate positive and negative examples (P and N), we can assign them to G and V. When we discover positive rules (people that are in a child relation), we use P as generation set G and N as validation set V. When we discover negative rules (people that are not in a child relation), we use N as generation set G and P as validation set V. Note that our approach is independent on how G and V are generated: they could also be manually generated by some domain experts, which would require additional manual effort.

4. A* GREEDY ALGORITHM

Since the universe of all possible rules R is too big to enumerate, we solve the online variant of the above problem. Paolo: if offline problem is NP, online is at least NP; to be verified Stefano: Shouldn't we just say that the problem is NP therefore we go for a greedy algorithm that allows also to avoid the enumeration of all rules?

4.1 Optimality

Define property on why the A* algorithm produces the greedy solution. Maybe study when the greedy solution become optimal? (If all rules identify disjoint set of input example, then greedy solution is optimal)

5. EXPERIMENTS

We carried out an extensive experimental evaluation of our rules discovery approach. We grouped the evaluation into 4 main sub-categories: *(i)* a first set of experiments aims at demonstrating the quality of our output, both for negative and positive rules; *(ii)* a second set of experiments compares our method with state-of-the-art systems; *(iii)* in the third set of experiments we outline the applicability of rules discovery by enhancing machine learning algorithms; *(iv)* in the last set of experiments we discuss internal system settings and some optimization techniques.

Settings. We evaluated our approach over several popular KBs. For each KB, we downloaded the most up-to-date core facts and loaded them into our SPARQL query engine. We experimented several SPARQL engines, including Jena ARQ, OWLIM Lite, and RDF-3x. We also implemented a naïve relational database solution with PostgreSQL. Eventually we opted for OpenLink Virtuoso, as it was the fastest among all the solutions. Virtuoso took on average 20 minutes to load a medium size KB (i.e., 10 GB) into its store, and around 100 milliseconds to excute a single hop query. All experiments are run on a iMac desktop with an Intel quad-core i7 at 3.40GHz with 16 GB RAM. We run Virtuoso server with its SPARQL query endpoint on the same machine, optimized for a 8 GB available RAM.

Our method needs the two input parameters α and β of Equation(REF.). α measures the importance of the coverage over the generation set, while β measures the coverage over the validation set. In other words, a high α priviliges recall over precision, while a high β gives more importance to precision. We can afford a high α and a low β when the input KB is accurate and complete. We will show that KBs contain many errors and missing information, therefore we set $\alpha = 0.3$ and $\beta = 0.7$. Increasing α and decreasing β means a higher number of output rules with a lower accuracy.

We also set the $MaxPathLen$ parameter to 3. This number represents the maximum number of atoms that we can have in the body of discovered rules. We will show in Section 5.4 that increasing this parameter does not bring any benefits, as body rules longer than 3 atoms start to be very complicated and not insightful.

Evaluation Metrics. Our approach discovers rules for a given target predicate. For each KB, we chose 5 representative predicates as follows: we first ordered predicates according to descending popularity (i.e., number of triples having that predicate), and then we picked the top 3 predicates for which we know there exists at least one meaningful rule, and other 2 top predicates for which we did not know whether some meaningful rules existed. We repeat the procedure for each input KB, and for positive and negative rules. Despite working one predicate at time, we can also discover rules for the entire KB by listing all predicates in the KB, and discover rules for each of them. We will show in Section 5.2 how this can be done.

Positive rules are very useful to enrich the KB by discovering new facts. Since we are generating new data, we cannot evaluate the induced rules over the existing data. Therefore we proceed as follows: we run the algorithm over the KB, and for each output rule we generate all new predictions

that are not already in the KB (we execute the head of the rule against the KB and we remove all those pairs that are already connected by the target predicate in the KB). As an example, for the rule $\texttt{spouse}(b, a) \Rightarrow \texttt{spouse}(a, b)$, we retrieve all the pairs (b, a) such that b is *spouse* with a but a is not *spouse* with b in the KB. If the rule is universally correct (like the previous example), we mark all the new predictions as true. If the rule is unknown, we randomly sampled 30 new predictions and manually check them against the Web. The *precision* of the rule is then computed as the ratio of true predictions out of true and false predictions.

Negative rules are slighlty more complicated to evaluate. In fact, despite KBs are usually incomplete, the majority of the data not stated in the KB is false – if we take all the people in a KB, a very small fraction of the cartesian product of all the people will be actually married. Therefore negative rules will always discover many correct negative facts. However, negative rules are a great means to discover errors in the KB, and we leverage this aspect to evaluate them. For each discovered rule, we retrieve from the KB pairs of entities for which the body of the rule can be instantiated and that are also connected by the target predicate. As an example, for the rule $\texttt{child}(a, b) \Rightarrow \neg\texttt{spouse}(a, b)$, we retrive all the pairs (a, b) such that b is *child* of a and a is *spouse* with b. We call these generated pairs of entities *potential errors*. Similarly to positive rules, whenever a rule is universally correct we mark all its potential errors as true, whereas if the rule is unknown we manually check 30 sampled potential errors. The final precision of a rule is computed as actual errors divided by all potential errors. Stefano: Shall we say that a pair of entities is a true error not only if the final pair is wrong, but also if some intermediate values are wrong? Like the birthYear and foundingYear for a founder relation, we consider a true error not only if the person is not the actual founder, but also if the birthYear and/or the foundingYear are wrong.

5.1 Rules Discovery Accuracy

The first set of experiments aims at evaluating the accuracy of discovered rules over the 3 most popular and widely used KBs: DBPEDIA [3], YAGO [11], and WIKIDATA [12]. For each KB we downloaded the most recent version and selected core facts, facts about people, geolocations and transitive `rdf:type` facts. WIKIDATA provides only the enitre dump, therefore we just eliminated from it no-english literal values. Table 1 shows the characteristics of the 3 KBs.

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

As the figure shows, the size of the KB is relevant. Loading the entire KB into main memory is not feasible unless we have high memory availability (cite the two SIGMOD16 papers), or we reduce the KB by eliminating facts such as `rdf:type` or literals [7]. We propose an approach that is disk-based where only a small portion of the KB is loaded into main memory, such that we can discover rules on any size KB with a normal machine.

Positive Rules Discovery. We first evaluate the precision of positive rules for the top 5 predicates on the 3 KBs. The number of new induced facts varies significantly from rule to rule – for instance a rule with literals comparison will produce a very high number of facts. In order to avoid the precision to be dominated by such rules, we first compute the precision for each rule as explained above, and then we average values over all induced rules. Table 2 reports precision values, along with predicates average running time.

Table 2: Positive Rules Accuracy.

KB	*Avg. Running Time*	*Precision*
DBPEDIA	34min, 56sec	**63.99**%
YAGO	59min, 25sec	**62.86**%
WIKIDATA	2h, 21min, 34sec	**73.33**%

Our first observation is that the more accurate is the KB, the better is the quality of induced rules. WIKIDATA contains very few errors, since it is manually curated and every triple is manually checked by different individuals before being inserted. DBPEDIA and YAGO instead are automatically generated by extracting information from the Web, hence their quality is significantly lower. Discovering perfect positive rules is a hard task, mostly because there is no guarantee of the existence of valid negative examples. A striking example in this direction is the rule induced for `founder` in DBPEDIA. Our approach discovers that if a person is born in the same place where a company is founded, then the person is the founder of the company. The rule is obviously wrong, as there are many people who are born in the same place of a company and have not founded the company. However this rule has a very high coverage over the generation set (many companies' founders founded their companies in their birth place), and a very low coverage over the validation set – indeed among the cartesian product of all the people and companies, a very small fraction includes people and companies born and founded in the same place. Despite such hard cases, our approach is always capable of producing correct rules for those predicates for which we knew there existed some valid rules. Cases like `academicAdvisor`, `child`, and `spouse` have a precision above 95% in all of the KBs, and final values are brought down by few predicates where meaningful rules probably do not exist at all.

The running time is influenced by different factors. First of all the size of the KB has obviously a huge impact, as we are slower in WIKIDATA which is the biggest KB. Not only the number of triples is relevant, but also the different number of predicates. In fact the more predicates we have in the KB, the more alternative paths we observe when traversing the graph, hence a bigger search space. The second relevant aspect is the target predicate involved. We noticed that some kind of entities have a huge number of outgoing and incoming edges (*United States* in WIKIDATA is connected to more than 600K entities). When the generation set includes such type of entities, the navigation of the graph is slower as we need to traverse a high number of edges. This is what happens in YAGO, where the most popular predicates are `isLeaderOf` and `exports`. Eventually the *maxPathLen* parameter also has a big say in the final running time. The longer the rule, the bigger is the search space. We will show in the next Section how we can be much faster if we set to 2 atoms the maximum length of the rule. Section 5.4 will dis-

cuss some optimization techniques to significantly cut down the running time based on the above observations.

Negative Rules Discovery. Negative rules are very useful to discover inconsistencies in the KB. We evaluated discovered 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 negative rules, whereas the precision is computed as the percentage of actual errors among potential errors.

Table 3: Negative Rules Accuracy.

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

Negative rules generally have better accuracy than positive ones. This is mostly due to the completeness of the validation set: for negative rules the validation is the universe of all possible counter examples stated in the KB, whereas for positive rules the validation set is just a small fraction of it. Therefore usually negative rules are better validated than positive ones. WIKIDATA shows lower numbers just because it does not contain as many errors as DBPEDIA and YAGO: even though discovered rules are almost correct, the percentage of actual errors identified is lower in WIKIDATA. As an examples, our approach identifies the same rule that two people with same gender cannot be married both in YAGO and WIKIDATA. Such rule identifies errors in YAGO with 94% accuracy, while the accuracy in WIKIDATA for the same rule is 57%. YAGO is the KB with the highest number of errors. As an example, there are 9,057 cases where a child is born before her parent. We cannot point exactly where the error is – there could be an error in one of the birth dates or an error in the parentship relation – but we can be sure that at least one of these values is wrong and we can send them to human evaluators to check exactly where the inconsistency is.

Differently from positive rules, literals play a vital role in discovering negative rules. In fact in many cases correct negative rules relies on temporal aspects in which 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 usually faster than discovering positive rules. This is mostly due to the time we spend executing validation queries. Whenever we discover a rule that respect our language bias (Section 2.1), 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 just all the entities connected by the target predicate, hence easier to compute by the SPARQL engine.

Stefano: Shall we show some meaningful rules here?

5.2 Comparative Evaluation

We compare the performance of our rules discovery method against AMIE [7], the state-of-the-art system in discovery Horn Rules from KBs.

AMIE is a rules discovery system designed to discover positive rules. It first loads the entire KB into memory, and then it discover positive rules for every predicate in the KB. AMIE lists all the predicates in the KB and insert each of them in the head of the rule. Once the head is filled, the system tries to expand the rule by pivoting on one of the variable of the current predicate and looking for predicates having the same variable with high coverage in the KB. The coverage of a rule is penalized with the partial closed world assumption, where the set of negative examples for a given pair (x, y) and a target predicate p is all those pairs where x is connected with p to another entity that is not y. Differently from us, AMIE outputs all possible rules that exceeds a given threshold and rank them according to their coverage function.

Given the in-memory implementation, AMIE cannot handle large KBs. We tried to run it on the KBs of Table 1, but the system goes quickly out of memory. Therefore we downloaded and used the modified versions of YAGO and DBPEDIA used in their experiments [1]. These versions consist in the core facts of the KB, without literals and `rdf:type` facts. Table 4 summarizes the characheristic of this dataset.

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 (Table 1). Since our approach needs the type information, 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 against AMIE on its natural setting: discovery of positive rules. AMIE takes as input an entire KB, and discover rules for every relation in the KB. We adapted our system to discover rules for every relation in the KB 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 types for both subject and object. The most common type is computed as the most popular `rdf:type` that is not super class of any other most popular type. After computing type domain and co-domain for each predicate, we run our approach sequentially on every predicate. Furthemore we set the $maxPathLen$ parameter to 2, since this is the default setting for AMIE.

AMIE outputs a huge amount of rules along with their score – 75 otuput rules in YAGO, and 6090 in DBPEDIA. We followed their experiments 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. Our approach is more conservative and produces muche less rules than AMIE. We noticed that for every predicate AMIE always produces more than one rule, while there are several cases where the output of our algorithm is empty since none of the plausible rules have enough support. This results, for instance, in just 11 rules in output on the entire YAGO. The precision of each rule is computed as explained above with a minor modification: whenever a rule is unknown with its new predictions, we first check the existence of the new induced facts in a newer

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

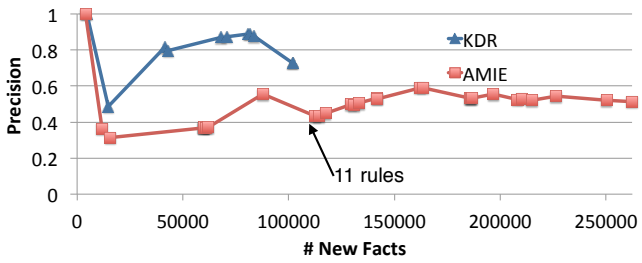

Figure 2: Predictions Accuracy on Yago

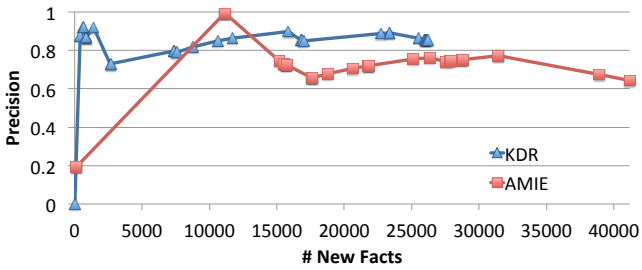

Figure 3: Predictions Accuracy on DBPedia

version of the KB. This is possible because the dataset does not contain the most up to date versions. If a new fact does not appear in the newer version and neither on the Web, is then evaluated as false.

Figure 2 plots the total number of new unique predictions (x-axis) versus the aggregated precision (y-axis). The n-th point from the left represents the total number of predictions and the total precision of these predictions, computed over the first n rules (sorted according to AMIE's score). AMIE produces many more predictions (262K vs 102K), but with a singnificant lower accuracy. This is due to the high number of rules in output of AMIE, but also in the way these rules are ranked. In fact if we limit the output of AMIE to 11 rules (same output of our approach), the final accuracy is still 29% below our approach, with just 10K more predictions. In fact many good rules are preceeded by meanignless rules in the ranked output, and it is not clear how to set a proper k in order to get the best top k rules. Our approach instead understands that in some cases meaningful rules do not exist, and it outputs something only when it has a strong confidence. This results in a lower number of predictions with a very high accuracy (precision is above 85% with 80K predictions). Moreover, our approach can also simulate AMIE if we are more interested in recall. If we modify the α andβ parameters, we can obtain a higher number of predictions in output, at the expense of accuracy.

Figure 3 shows the same evaluation on DBPedia. DBPedia has a richer set of relations, therefore also our approach is capable of producing 30 rules in output. Despite the same number of rules, once again our approach leads to a lower number of predictions (26K vs 41K) with a singificant higher accuracy (85% vs 74%). The only point where AMIE can outpeform our approach is when we consider the top 3 rules: the third rule discovered by AMIE is indeed a universally true rule that produces more than 11K correct predictions.

Negative Rules. As a second set of comparative experiments we used AMIE 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. Therefore we proceed as follows: we sampled the top 5 most popular predicates in each KB and we created for each predicate a set of negative examples as explained in Section 3.2. For each negative example we added to the KB a new fact connecting the two entities with the *negation* of the predicate. For example, we added a `notSpouse` predicate connecting each pair of people who we believe are not married according to our negative examples generation technique. We then run AMIE only on these new created predicates. The evaluation of negative rules is then carried out as before: we generate potential errors in the KB, and we manually evaluate the precision of such errors. Table 5 shows the results on the two KBs.

Table 5: Negative Rules vs AMIE.

	AMIE		KRD	
KB	*# Errors*	*Precision*	*# Errors*	*Precision*
DBPedia	457	38.85%	148	**57.76%**
Yago	633	48.81%	550	**68.73%**

Our approach outperforms AMIE in both cases of almost 20%. This is because for the negation of a predicate, we use the actual predicate as counter examples. AMIE instead is looking only at the negation of the predicate, hence it is much less precise. In fact the output of AMIE consists in a high number of rules for each predicate (often more than 30), and in many cases AMIE produces same rules for both the positive and negative scenario. As an examples, AMIE outputs that if a country a exports a good b, then a imports b and a does not import b.

Despite clearly outperforming AMIE, numbers look signficnatly lower that the ones showed in Section 5.1. This is because in this experiment we are using the AMIE modified KBs which do not contain literals. As previously mentioned, literals play a vital role when discovering negative rules, both in terms of total errors discovered and in terms of precision. Excluding literals is a big disadvntage, and we will show this aspect in details in Section 5.4.

Running Time. We report here the running time of AMIE and our approach on our Desktop machine. Note that numbers here are different from [7], where AMIE was run on a 48 GB RAM server. AMIE could finish the computation only on Yago 2, while for the other KB it gets stuck without outputting any more rules. When this happened, we stopped the computation after we did not see any new rules in output for more than 2 hours.

Table 6: Run Time vs AMIE.

KB	*#Triple*	*#Predicates*	*AMIE*	KRD	*Types Time*
Yago 2	948.3K	20	30s	18m,15s	12s
Yago 2s	4.1M	26 (38)	>8h	47m,10s	11s
DBPedia 2.0	7M	904 (10342)	>10h	7h,12m	77s
DBPedia 3.8	11M	237 (649)	>15h	8h,10m	37s
Wikidata	8.4M	118 (430)	>25h	8h,2m	11s
Yago 3*	88.3M	72	-	2h,35m	128s

Table 6 reports the running time on different KBs. The first five KBs are the AMIE modified version, while Yago 3* is complete Yago, including literals and `rdf:type`. The

third column shows the number of predicates for which AMIE was able to produce at least one rule. In some cases AMIE got stuck without producing any rules for some predicates, hence we report the total number of predicates in brackests. For a fair comparison we run our algorithm only on those predicates for which AMIE could prodcue at least one rule. The foruth and fifth columns report the total running time of the two approaches. Despite being disk-based, our approach can successfully complete the task faster than AMIE, except the case of YAGO 2. This is because of the very small size of the KB, which can easily fit in main memory. However, when we deal we real KBs (YAGO 3*), AMIE is not even capable of loading the KB due to out of memory errors. Eventually the last column reports the total time needed to compute `rdf:type` information for each predicate in the KB. Such time is negligible w.r.t. the total running time. The running time justify the disk-based strategy: our approach can succesfully discover rules for any size KBs on any machine.

Other Systems. We found other available systems to discover rules in KBs. [1] discover new facts at instance level, hence less generic than our approach. On AMIE YAGO 2 KB they can discover 2K new facts with a precision lower than 70%. The best rule we discover on YAGO 2 already discovers more than 2K facts with a 100% precision. Ontological Path Finding (TO CITE) implements AMIE algorithm with a focus on scalability. They do not introduce any novelty in the algorithmic part, but just a clever way of splitting the KB into multiple cluster nodes so that the computation can be run in parallel. The output is the same as AMIE. Eventually we did not compare with classic Inductive Logic Programming systesm [5,8], as these are already significantly outperformed by AMIE both in accuracy and running time.

5.3 Machine Learning Application

The main goal of this set of experiments is to prove the applicability of our approach in helping Machine Learning algorithms to provide meaningful training examples. We chose DeepDive [10], a Machine Learning approach to incrementally construct KBs. DeepDive extracts entities and relations among entities from text articles via distant supervision. The key idea in distant supervision is to use an external source of information (e.g., a KB) to provide training examples for a supervised algorithm. As an example, the main showcase in DeepDive extracts mentions of married people from text documents. In such scenario DeepDive uses as a first step DBPEDIA in order to label some pairs of entities as *true* positive (those pairs of entities that can be found in DBPEDIA). These labelled examples are then used to construct a factor graph, similar to Markov Logic, that will be used to predict labels on the rest of the candidates. Unfortunately, a KB can provide only positive examples. Hence in DeepDive the burden of creating negative examples is left to the user thorugh rules definition.

In this set of experiments we will use our negative rules on DBPEDIA to generate negative examples, and compare the output of DeepDive generated with our negative examples with the manually defined negative examples. We used DeepDive showcase example[2], where the goal is to extract mentions of married people from text articles. DeepDive already provides some negative rules to generate negative examples (e.g., if two people appear in a sentence connected

[2] `http://deepdive.stanford.edu/`

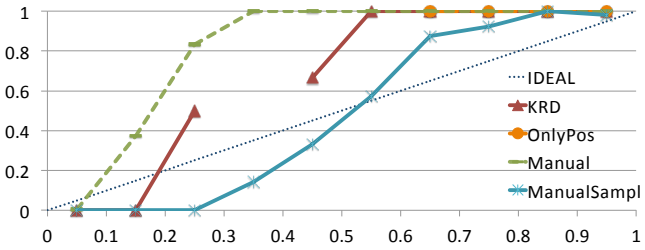

Figure 4: DeepDive Application 1K articles

by the word *brother* or *sister* then they are not married). We therefore compare the output of DeepDive using our generated negative examples and the ones generated by DeepDive rules.

Figure 5 shows DeepDive accuracy plot run on 1K documents as input. The accuracy plot shows the ratio of correct positive predictions over positive and negative predications (y-axis), for each probability output value (x-axis). The dotted blue line represents the ideal situation, where the system finds high number of evidence positive predictions for higher probability output values (when the output probability is 0 there should not be positive predictions). The plot is computed over a test set, while the system is trained over a separated training set. The Figure shows 4 lines other than the ideal ones. KRD 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 negative examples in size equal to positive examples. The first observation is that `OnlyPos` and `Manual` do not provide a 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 mostly the absence of data: over the input 1K articles, we could find only 20 positive and 15 negative examples from DBPEDIA.

If we extend the input to 1M articles, things change drastically (Figure 5). All the three approaches except `OnlyPos` can correctly drive DeepDive in the training, with the examples provided with our technique leading to a slightly better result. This is because of the quality of the negative examples: our negative rules generate significant examples that can help DeepDive in understanding discriminatory features between positive and negative labels. The output of `ManualSampl` and KRD are very similar, meaning that we can use our approach to simulate user's behaviour and provide

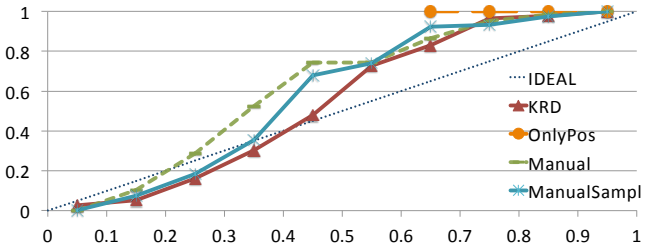

Figure 5: DeepDive Application 1M articles

negative examples. With manually defined rules we generate a higher number of examples (23K against 5K generated from DBPedia), however the results are very similar since a small number of significant examples is enough to provide complete evidence for the training. As long as we have an external source of information with a decent coverage over the input articles, users do not need to worry to provide rules to generate negative examples.

5.4 Ablation Study

Optimization Techniques. Limit on incoming and outgoing edges, samples of generation set: random and smart sampling.

Literals Effect. Remove literals from the KB and see what happens.

Rules Length. Try lenght 2 and 4.

6. RELATED WORK

AMIE [7].

[13] – Jiawei's work, discover of future autorship relations from DBLP (Vamsi?).

[1] – induction of new facts at instance level, rather than being generic and induce rules with variables. Discover much less new instances (2K vs our 100K on Yago2), and even the precision does not look great. Probably not worth to compare against them, just mention.

ALEPH [8], WARMR [5], and Sherlock [9] – Inductive Logic Programming approaches (outperformed by AMIE).

Yago [11], DBPedia [3], and Wikidata [12].

Association Rules Mining [2]: given a database of costumer transactions, generate associtation rules that correlates items in the database (e.g., customers who usually buy bread and butter also purchase milk). Different setting (search space small), and usually confidence thresholds are difficult to set. Cannot be applied to KBs because building a relational database from a graph means compute all possible instantiations of a given relation, which makes the search space too big (this is why traditional databases algorithms are difficult to apply over KBs).

"Ontological Pathfinding: Mining First-Order Knowledge from Large Knowledge Bases." (SIGMOD '16, to be published). Same language and similar algorithm of AMIE, they just focus on scalability (they load the DB in a relational db and split it in different batches, to apply rules discovery in each batch independently and merge the results in a map-reduce fashion). Still require many resources (different machines and/or cores), and they do not consider literals.

"CLAMS: Bringing Quality to Data Lakes" (SIGMOD DEMO '16, to be published). Discover conditional denial constraints on RDF data. Constraint is made by a conjuctive query and a denial constraint over the conjcutive query. Example: query = select all locations t1, t2 that have a latitude and are connected by sameAs relation; constraint = for every t1, t2 from the query, it cannot exist t1.latitude != t2.latitude. In the demo paper they do not explain how they find such constraints, but they probably adapt state of the art algorithm for denial constraints. After discovering constraints they discover vilation, rank them and let the user solve such violations. All run on Sparq framework with 10 worker nodes with 832G Ram (everything loaded in memory).

7. CONCLUSION

8. REFERENCES

[1] Z. Abedjan and F. Naumann. Amending rdf entities with new facts. In *The Semantic Web: ESWC 2014 Satellite Events*, pages 131–143. Springer, 2014.

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

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

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

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

[6] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *SIGKDD*, pages 601–610. ACM, 2014.

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

[8] S. Muggleton. Inverse entailment and progol. *New generation computing*, 13(3-4):245–286, 1995.

[9] S. Schoenmackers, O. Etzioni, D. S. Weld, and J. Davis. Learning first-order horn clauses from web text. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1088–1098. Association for Computational Linguistics, 2010.

[10] J. Shin, S. Wu, F. Wang, C. De Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using deepdive. *Proceedings of the VLDB Endowment*, 8(11):1310–1321, 2015.

[11] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706. ACM, 2007.

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

[13] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu. Mining top-k large structural patterns in a massive network. *PVLDB*, 4(11):807–818, 2011.