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, 6]): 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 [3], 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 [6](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.

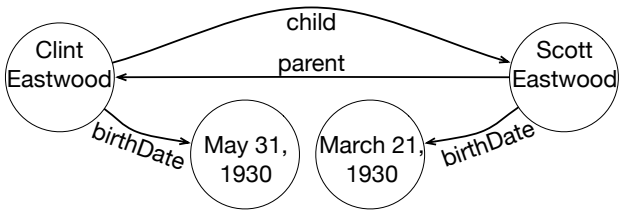

Figure 1: Graph Portion of DBPedia

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*) [5, 6]. 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 [6]).

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

5.1 Negative Rules Evaluation

Evaluation of negative rules.

5.2 Comparison Evaluation

Comparison against AMIE and evaluation of positive rules.

5.3 Machine Learning Application

DeepDive.

6. RELATED WORK

AMIE [6].

[11] – 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 [7], WARMR [4], and Sherlock [8] – Inductive Logic Programming approaches (outperformed by AMIE).

YAGO [9], DBPEDIA [2], and WIKIDATA [10].

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] 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.

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] 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.