Machine Learning (2019) 108:1165–1189
https://doi.org/10.1007/s10994-019-05798-y

Online probabilistic theory revision from examples with ProPPR

Victor Guimarães[1] · **Aline Paes**[2] · **Gerson Zaverucha**[1]

Received: 29 December 2017 / Accepted: 20 April 2019 / Published online: 14 May 2019

Abstract
Handling relational data streams has become a crucial task, given the availability of pervasive sensors and Internet-produced content, such as social networks and knowledge graphs. In a relational environment, this is a particularly challenging task, since one cannot assure that the streams of examples are independent along the iterations. Thus, most relational learning systems are still designed to learn only from closed batches of data. Furthermore, in case there is a previously acquired model, these systems either would discard it or assuming it as correct. In this work, we propose an online relational learning algorithm that can handle continuous, open-ended streams of relational examples as they arrive. We employ techniques of theory revision to take advantage of the previously acquired model as a starting point, by finding where it should be modified to cope with the new examples, and automatically update it. We rely on the Hoeffding's bound statistical theory to decide if the model must, in fact, be updated in accordance with the new examples. The proposed algorithm is built upon ProPPR statistical relational language, aiming at contemplating the uncertainty inherent to real data. Experimental results in social networks and entity co-reference datasets show the potential of the proposed approach compared to other relational learners.

Keywords Online learning · Statistical relational learning · Theory revision from examples · Inductive logic programming · Mining data streams

Editors: Nicolas Lachiche, Christel Vrain, Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese.

✉ Aline Paes
alinepaes@ic.uff.br

Victor Guimarães
guimaraes@cos.ufrj.br

Gerson Zaverucha
gerson@cos.ufrj.br

1 Department of Systems Engineering and Computer Science, COPPE, Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, RJ, Brazil

2 Department of Computer Science, Universidade Federal Fluminense (UFF), Niterói, RJ, Brazil

1 Introduction

The availability of sources that continuously generate data has aroused the need for specific Machine Learning (Mitchell 1997) methods that are capable of processing such data incrementally. Regular batch methods would have difficulties on coping with such dynamic streams as they arrive while still considering the previously acquired models. Thus, several online algorithms that learn from continuous, open-ended data streams have been proposed in the last decades (Domingos and Hulten 2000; Bifet et al. 2010; Dries and De Raedt 2010; Gama and Kosina 2011).

Most of the stream mining methods are designed to deal with propositional data only, in the sense that the examples are independent and homogeneously distributed, and therefore characterized in an attribute-value format with no explicit relationship with each other. However, real-world data that also present the continuous arrival behavior, such as drug discovery (Tsunoyama et al. 2008), social networks (Getoor and Diehl 2005) and knowledge bases and graphs (Craven et al. 2000; Pujara et al. 2015), are heterogeneous, multi-related, uncertain and noisy.

Thus, although there exist languages and algorithms to learn from them that are capable of expressing concepts, objects, and their relationships, they are not designed to handle data streams. Such methods, which aggregate techniques from Machine Learning, Knowledge Representation and Reasoning under Uncertainty, compose the area of Statistical Relational AI (StarAI) (De Raedt et al. 2016). Two main components compound the StarAI languages: a qualitative one, usually represented by a rich and expressive structure such as first-order logic, and a quantitative one, mostly characterized by probabilistic parameters. To learn the structure, a number of StarAI methods take advantage of Inductive Logic Programming (Muggleton and De Raedt 1994), while addressing contributions from statistical and probabilistic models (Murphy 2012) to learn the parameters.

Learning in such languages brings the advantage of inducing an interpretable and explainable model, from a set of examples and even regarding *Background Knowledge* (BK). The possibility of considering a BK may considerably boost the learning process from data streams, as existing learned models, induced from a previous incoming of examples, can pose as the BK.

However, the majority of StarAI learning algorithms assume that the BK is fixed and correct, and therefore not changeable. Notwithstanding, it may be the case that part of the BK is incorrect or even incomplete. This is particularly true when the BK has been previously acquired from an old set of examples, as happens in data streams environments. In such cases, to make the BK useful to the final model, it is necessary that it undergoes a data-oriented modification process. Modifying an initial knowledge from a set of examples is precisely the goal of *theory revision from examples* (Richards and Mooney 1995; Paes et al. 2005; Duboc et al. 2009; Paes et al. 2017) algorithms. Besides, such methods are also capable of inducing new rules, given the examples and BK. Also, previous works have shown that theory revision could improve the quality of learned models, while using fewer examples than the purely inductive methods (Richards and Mooney 1995; Duboc et al. 2009; Paes et al. 2017).

Thus, an online relational learning environment may benefit from theory revision to take the previously learned model as a starting point, check where it should be modified, and change it, improving its quality when facing new examples.

In this paper, motivated by the need of continuously learning from relational, uncertain, and open-ended data, we contribute with an algorithm that can learn and revise a StarAI model online, named OSLR (Online Structure Learner by Revision).[1] We built the system to learn models represented with ProPPR language (Wang et al. 2015) since it can efficiently handle inference on noisy relational data.

Nevertheless, a question that still arises is when to decide that the existing model needs to be revised. Trying to correctly change the model considering *each* new example that it is not generalized yet could quickly lead to overfitting and impose high costs on the learning process. Thus, in this work, we use the Hoeffding's bound (Hoeffding 1963; Domingos and Hulten 2000; Cardoso and Zaverucha 2006; Lopes and Zaverucha 2009; Srinivasan and Bain 2017) to decide whether a revision improvement on the existing model is significant so that it should be implemented.

Furthermore, to avoid the bias caused by statistical dependences between relational instances (Jensen and Neville 2002), particularly because the Hoeffding's bound was originally developed to handle i.i.d. cases, we take extra care to regard the linkage between instances.

We compare our proposed algorithm against a batch StarAI learning method, namely the RDN-Boost system, designed to learn Relational Dependency Network (RDN) (Natarajan et al. 2012; Khot et al. 2015), that has obtained the state-of-the-art results in a number of relational datasets. We designed a simulated online environment to emulate what would happen in the real world when new examples would appear incrementally. Additionally, we also present competitive results against HTilde (Lopes and Zaverucha 2009; Menezes 2011), which is an online algorithm devised to handle streaming relational data but assuming that they are all independent and free of uncertainty.

The experiments demonstrated that our proposal is promising in the online environment, especially in the initial iterations, when only a few examples are available to train the model.

The remainder of the paper is organized as follows: we present the works related to ours in Sect. 2. In Sect. 3 we give the background knowledge to understand our work. We present our proposal in Sect. 4 followed by the performed experiments in Sect. 5. Finally, we conclude and give some directions for future work in Sect. 6.

2 Related work

We are using techniques from theory revision from examples to improve the quality of the StarAI model for a new set of examples. A well-known theory revision system is FORTE (Richards and Mooney 1995). FORTE revises first-order logical theories by first finding the points where the theory is misclassifying the examples (the *revision points*) and then, applying *revision operators* to these points. We also follow this key theory revision top-level procedure here.

With FORTE's successor, FORTE-MBC (Duboc et al. 2009), we share the idea of using a Bottom Clause (Muggleton 1995) to support the generation of new literals. However, we do not employ stochastic search here as the FORTE-MBC successor, YAVFORTE (Paes et al. 2017). While FORTE and its successors aim at revising theories considering a whole set of examples at "one-shot", i.e., they are essentially batch algorithms, here we take advantage of revision techniques to allow learning first-order models in an online environment.

To handle examples coming in streams and revise theories in an online fashion, we have to decide when to apply a revision given the examples. To take such a decision, we use the Hoeffding's bound (Hoeffding 1963). The Hoeffding's bound has already been used

[1] The OSLR system is publicly available at https://github.com/guimaraes13/oslr.

in a classical Machine Learning method, yielding the VFDT (Domingos and Hulten 2000) algorithm, which is a system to learning decision trees incrementally, by applying the Hoeffding's bound to decide whether to split a node, based on the information gain of the examples. Also, the Hoeffding's bound has been applied to learn massive data streams from relational databases (Hulten et al. 2003). However, in both cases, the bound is employed over i.i.d data, as in the second case the data is propositionalized before the statistical test is applied.

Another algorithm related to ours is $HTilde$, introduced in Lopes and Zaverucha (2009). In Menezes (2011) this algorithm was enhanced, and it generated new results. HTilde was implemented over Tilde (Blockeel and De Raedt 1998; Blockeel et al. 1999) to incrementally learn first-order logical decision trees by using the Hoeffding's bound.

The main difference between our approach and HTilde is that HTilde can only *grow* the first-order tree model, which may produce quite complicated final trees. Our approach, on the other hand, can also to create new rules, and add literals to existing rules, besides deleting entire rules, and literals from existing rules, according to the incoming of new examples. As our approach allows more deep changes in the theory, while still regarding the useful information in the previous model, it needs fewer examples to perform well, compared to HTilde.

Another benefit of our approach over HTilde is that we rely on a StarAI system. In this way, we can handle noise and uncertainty inherent to real-world examples, while HTilde is based on a purely logical language.

As the model language and inference engine we use ProPPR (Wang et al. 2015) that is a StarAI system based on an approximation of the PageRank algorithm (Page et al. 1999) to find a probabilistic distribution over the possible answers of a logical query. ProPPR is capable of using a set of positive and negative examples to tune this distribution in order to increase the weight of desired answers and decrease the weight of undesired ones. We have chosen to rely on ProPPR instead of other StarAI languages such as Markov Logic Networks (Richardson and Domingos 2006) or ProbLog (De Raedt et al. 2007) because of its efficiency on both answering the logic queries and learning the weight of the desired answers.

3 Background knowledge

As stated before, we are using techniques from theory revision from examples, or simply theory revision, upon a probabilistic logic system called ProPPR (Wang et al. 2015). In this section, first, we explain ProPPR's language and its inference engine. Then, we describe the key concepts related to theory revision (De Raedt 2008; Wrobel 2013).

3.1 ProPPR

ProPPR is a statistical relational system that uses a first-order probabilistic language, of the same name, to infer facts, given background knowledge and a set of definite clauses (theory) (Wang et al. 2015).

3.1.1 Language

ProPPR follows the language of function-free first-order logic with the main syntactic elements defined as follows: a *variable* is represented by a string of letters, digits or underscores, starting with an upper case letter. A *constant* is represented like a variable, but starting with

Table 1 ProPPR language example

(1)	$about(P, L) \leftarrow hasWord(P, L) \wedge isLabel(L)\ \underline{\{w1\}}$.
(2)	$about(P, L) \leftarrow linksTo(P, P1) \wedge hasLabel(P1, L)\ \underline{\wedge\ weight2(P1, L)\ \{w2\}}$.
(3)	$about(P, L) \leftarrow linksTo(P, P1) \wedge hasWord(P1, L) \wedge isLabel(L)\ \underline{\wedge\ weight3(P1, L)\ \{w3\}}$.
(4)	$\underline{weight2(P1, L) \leftarrow \{w2(P1, L)\}}$.
(5)	$\underline{weight3(P1, L) \leftarrow \{w3(P1, L)\}.}$

a lower case letter. A *term* is either a constant or a variable. A predicate is represented by a string of letters, digits or underscores, starting with a lower case letter. An *atom* is a predicate possibly followed by a n-tuple of terms between brackets; in this case, we say the predicate has arity n.

A knowledge base written in ProPPR's language can be composed of ground atoms (facts) and definite clauses. A definite clause is as follows:

$$H \leftarrow B_1 \wedge \cdots \wedge B_n.$$

where H and B_i are first-order logic atoms. H is called the head of the clause and $B_1 \wedge \cdots \wedge B_n$ is the body. The clause states that whenever the body is true, the head must also be true. If $n = 0$, we assume that the body is true. A definite clause is also called a rule, and the atoms in the body are called antecedents of the rule.

Features In addition to the logic part, ProPPR language allows us to specify features for the rules. The features are in the form of predicates and, when defined, they appear at the end of the rule between curly braces. The role of those features is related to the weight parameters. This is going to be clearer in the next two sections, where we explain the ProPPR's inference method and parameter learning algorithm. Table 1 shows an example of a set of rules in the ProPPR language. The underlined part has no logical utility; it is only there to define the features. Nonetheless, we see there that ProPPR accepts both first-order features, as in rules (2)–(5), and propositional ones, as in the first rule.

3.1.2 Inference

ProPPR defines logic inference as a search on a graph, similar to the one produced by a Prolog interpreter (Warren et al. 1977). Let P be a logic program that contains a set of rules $C = \{c_1, \ldots, c_n\}$. Let the query Q be a conjunction of predicates that appear in P. The graph is recursively constructed as follows: let v_0 be the "root" of the tree, representing the pair (Q, Q) and add it to an empty graph G. The root v_0 is labeled with Q. Next, add new nodes to G such as: let $(Q, (R_1, \ldots, R_k))$ be represented by a node u labeled with $R_1, \ldots, R_k$, and $c \in C$ be a rule of the form $R' \leftarrow S'_1, \ldots S'_l$, such that the pair (R_1, R') has the *most general unifier* $\theta = mgu(R1, R')$; then add a new edge $u \rightarrow v$ in G, where $v = (S'_1, \ldots S'_l, R_2, \ldots, R_k)\theta$. The edge $u \rightarrow v$ is annotated with (1) either R' or $R1$, in case R1 is solved, (2) the θ-substitution, when applicable, and (3) the feature associated with R' or with $R_1\theta$. The $(S'_1, \ldots S'_l, R_2, \ldots, R_k)$ is the associated subgoal list; an empty subgoal list represents a solution.

Note that in this formulation, the nodes are conjunctions of atoms, and the structure is, often, a directed graph, instead of a tree.

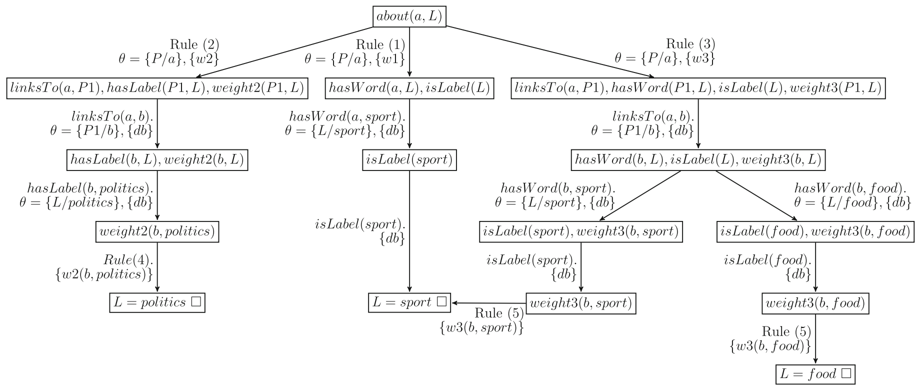

$about(a, L)$
Rule (2) $\theta = \{P/a\}, \{w2\}$
Rule (1) $\theta = \{P/a\}, \{w1\}$
Rule (3) $\theta = \{P/a\}, \{w3\}$
$linksTo(a, P1), hasLabel(P1, L), weight2(P1, L)$
$hasWord(a, L), isLabel(L)$
$linksTo(a, P1), hasWord(P1, L), isLabel(L), weight3(P1, L)$
$linksTo(a, b).$ $\theta = \{P1/b\}, \{db\}$
$hasWord(a, sport).$ $\theta = \{L/sport\}, \{db\}$
$linksTo(a, b).$ $\theta = \{P1/b\}, \{db\}$
$hasLabel(b, L), weight2(b, L)$
$isLabel(sport)$
$hasWord(b, L), isLabel(L), weight3(b, L)$
$hasLabel(b, politics).$ $\theta = \{L/politics\}, \{db\}$
$hasWord(b, sport).$ $\theta = \{L/sport\}, \{db\}$
$hasWord(b, food).$ $\theta = \{L/food\}, \{db\}$
$weight2(b, politics)$
$isLabel(sport).$ $\{db\}$
$isLabel(sport), weight3(b, sport)$
$isLabel(food), weight3(b, food)$
$Rule(4).$ $\{w2(b, politics)\}$
$isLabel(sport).$ $\{db\}$
$isLabel(food).$ $\{db\}$
$L = politics$ □
$L = sport$ □
$weight3(b, sport)$
$weight3(b, food)$
Rule (5) $\{w3(b, sport)\}$
Rule (5) $\{w3(b, food)\}$
$L = food$ □

Fig. 1 An example of ProPPR's resolution graph

Table 2 ProPPR facts example

$isLabel(sport).$	$hasLabel(b, politics).$
$isLabel(food).$	$hasWord(b, food).$
$linksTo(a, b).$	$hasWord(b, sport).$
$hasWord(a, sport).$	

Figure 1 shows part of the graph constructed by ProPPR given the query $about(a, L)$, the set of facts described in Table 2, and the program defined in Table 1. The nodes with a □ represent the solutions to the query. After constructing the graph, the answer to a query is found by an approximation of the Personalized PageRank algorithm, performed on the graph described above, via an algorithm called PageRank-Nibble (Andersen et al. 2006, 2007). This algorithm is used to implement a probabilistic SLD-Resolution and is outlined as Algorithm 1.

Algorithm 1 The PageRank-Nibble Algorithm

```
function PageRank-Nibble($v_0, \alpha', \epsilon$)
    $\mathbf{p} \leftarrow \mathbf{0}, \mathbf{r} \leftarrow \mathbf{0}, \mathbf{r}[v_0] \leftarrow 1, \hat{G} \leftarrow \emptyset$
    while $\exists u : \mathbf{r}(u)/|N(u)| > \epsilon$ do
        push($u$)
    end while
return p
end function

function push($u$)                                  ▷ this function modifies p, r and $\hat{G}$
    $\mathbf{p}[u] \leftarrow \mathbf{p}[u] + \alpha' * \mathbf{r}[u]$
    $\mathbf{r}[u] \leftarrow \mathbf{r}[u] * (1 - \alpha')$
    for $v \in N(u)$ do
        add edge $(u, v)$ to $\hat{G}$
        if $v = v_0$ then
            $\mathbf{r}[v] \leftarrow \mathbf{r}[v] + Pr(v|u) * \mathbf{r}[u]$
        else
            $\mathbf{r}[v] \leftarrow \mathbf{r}[v] + (Pr(v|u) - \alpha') * \mathbf{r}[u]$
        end if
    end for
end function
```

The algorithm works by maintaining two vectors **p** and **r** with the same size as the number of nodes in G, where **p** represents the PageRank approximation, and **r** represents a residual error of the approximation, for each node of G. Both vectors starts with 0 in each position, except for $r[v_0]$ that starts as 1, where v_0 represents the query that we would like to answer. Then, the algorithm repeatedly picks a node v with a large residual error and reduces this error by passing a fraction α' to $p[v]$. The remaining fraction is passed to $r[v_1]\ldots r[v_n]$, based on $Pr(v_i|v)$, that represents the probability of going from node v to v_i, and $v_i \in N(v)$ represents the neighbors of v. Finally, the probability of a solution s is $p[s]$, normalized by the sum of $p[s_i]$ for each solution s_i.

Depending on the program P and the set of facts, this graph might be very large or even infinite, so it is not fully constructed. ProPPR constructs the graph incrementally as necessary, depending on the values of α', ϵ and a predefined maximum depth.

In addition to the normal edges of the graph, ProPPR also adds two types of edges, namely: (1) an edge from every node to the starting node v_0, and (2) an edge from each node to itself. In the first case, those additional edges are used as bias, to give higher weights to shorter proofs. In the second case, it aims at making the solution nodes have higher probabilities than the other ones.

The PageRank procedure represents a random walk in the graph. Since each step has a probability of going back to the starting vertex, it is also known as a random walk with restart (Tong et al. 2006). A random walk represents the probability of, starting from a node v, reaching a node u, after a long time, by walking in the graph following a random edge at a time. Intuitively, as many paths from the root to a solution node there are, the higher is the probability of reaching this solution node, since there are more ways to get to it. Also, since every node has an edge to the starting node, ProPPR is biased to solutions that are closer to this later, representing solutions that are easier to explain.

3.1.3 Learning parameters

To learn the parameters of a model, as usual in Machine Learning, ProPPR requires examples. To cope with a relational model, where instances may be dependent, ProPPR defines an example as composed of a query of the type $p(X_1, \ldots, X_n)$, where p is a predicate, $n \geq 1$, and $X_i \in X_1, \ldots, X_n$ is either a constant or a variable, and at least one term $X_i \in X_1, \ldots, X_n$ must be a variable. The query is followed by a set of its possible instantiations (where all the variables are replaced by constants). Each possible instantiation is preceded by a + or a − sign, indicating whether it is a positive or a negative *answer*, respectively, where a negative answer represents an undesirable instantiation. An example is illustrated below:

$$about(a, X) \;\; +about(a, sport) \;\; +about(a, politics) \;\; -about(a, food)$$

The first literal is the query and the others represent possible answers.

Starting from the query, every time ProPPR adds a new edge to the resolution graph it marks such an edge with a feature. In case the edge has originated from a rule that has a feature in its body, the edge is annotated with that feature. When the edge has come from a fact, the edge is associated with a special feature, called db. If the rule has no related feature, ProPPR creates an unique feature for it. For instance, as we can see in Fig. 1, when ProPPR used Rule (2) to open the leftmost child of the root, it associated the feature $w2$ to the edge connection the root to this node. Analogously, when ProPPR used a the fact $linksTo(a, b)$ to continue the prove of this path, it associated the feature db for the new edge. These features are used to change the probability $Pr(v|u)$, tuning the weights of the answers. In this way, ProPPR can increase the weight of the desired answers and reduce the weights of the undesired ones, based on the examples. Thus, $Pr(v|u) \propto f(w_{u \rightarrow v})$ where $w_{u \rightarrow v}$ is a learned weight for the feature from the edge $u \rightarrow v$, and f is a differentiable function, by default, the exponential function.

To tune the weights, ProPPR generates the graph to answer each one of the examples and uses Stochastic Gradient Descent (SGD) to minimize the cross-entropy function, based on the proved answers. We refer the reader to Wang et al. (2015) for more details about ProPPR.

3.2 Theory revision from examples

The field of *theory revision from examples* focuses on modifying an initial theory to make it better suited to explain the set of examples (De Raedt 2008; Shapiro 1983). Theory revision methods are suitable to be applied in online environments, since they assume that in the presence of an initial model, it is better to start from it and modify it whenever the examples pointed out that this is necessary, instead of discarding it, or assuming it is correct.

Generally speaking, a theory revision top-level procedure is constituted of the following steps: (1) finding the examples incorrectly classified by the current model; (2) finding the points in the theory that are responsible for the model incorrectly classifying those examples, namely the *revision points*; (3) suggesting a proper change to these points, using *revision operators*; (4) choosing one of the proposed revisions and deciding whether or not it should be in fact implemented. Next, we briefly explain the concepts of *revision points* and *revision operators*.

3.2.1 Revision points

Revision points are rules and literals in a theory responsible for the misclassification of some example. Usually, there are two types of revision points:

– **Specialization revision point** the rules in the theory that take place in the proof of negative examples, indicating that the theory is too generic and needs to be specialized to avoid such proofs;
– **Generalization revision point** the literals in rules that prevent a positive example to be proved, thus indicating that the theory is too specific and needs to be generalized to allow positive examples to be proved.

In a single iteration of the search process of a theory revision system, it starts by identifying the misclassified examples, followed by the discovery of the revision points in the current theory, given a set of examples, to finally apply the respective revision operators to these points.

3.2.2 Revision operators

The four most common theory revision operators are: (1) *add rule*, that inserts a rule into the theory aiming at proving positive examples, either by copying an existing rule and modifying it by deleting and/or adding atoms, or creating a new one from scratch; (2) *delete antecedent*, that erases literals preventing positive examples being proved from the body of a rule; (3) *delete rule*, that erases a rule taking place in the proof of a negative example; and (4) *add antecedent*, that includes literals in the body of a rule, so that it becomes more specific and hence do not incorrectly proves examples. The first two operators aim at generalizing the theory, and the last two aim at specializing it.

Note that when applying a revision operator, it is possible that a former correctly classified example becomes misclassified. For example, when creating a rule, it is possible that a not covered negative example becomes provable. Therefore, the search procedure must decide which operator is going to bring more benefits to improve the theory, if any. Thus, normally, each revision operator is tried at each revision point, and the revision with the best score is kept. Then, the algorithm proceeds to find new revision points, and employ revisions on it, in a hill-climbing manner until no further improvements on the score are possible (Richards and Mooney 1995).

4 OSLR: online learning of ProPPR models based on theory revision

In this section, we present the algorithm proposed in this work, named Online Structure Learner by Revision (OSLR), which learns a ProPPR model online, i.e., from a stream of examples. We rely on theory revision techniques to address the necessary adaptations in the learned model, due to the continuously, open-ended incoming of examples.

OSLR top-level procedure is described as Algorithm 2. It represents what happens in a time instant of the data stream process, in a lifelong fashion, upon the arrival of new examples. The input has the following elements. First, we have the current model, which is either empty or an existing ProPPR model. In the last case, it may have been either induced in a previous iteration or elicited by an expert of the domain. To facilitate the proposal and implementation of possible revisions, in this work we adopt a tree to compactly represent the *logical* part of the model. We provide more details about it later in this section. Such a tree is updated in accordance with the implemented modifications, and, therefore, it is also received as an input to the algorithm. There are two possibilities for this variable: either it is bounded with the output from a previous iteration of the lifelong process, or it is created upon the first iteration of the algorithm. In this last case, it is created either with only the root, in case the very first initial theory is empty, or following a non-empty initial theory that could be elicited by an expert of the domain. Note that while thy has weights and features associated with the clauses, $tree$ holds only the clauses themselves.

The next input is the background knowledge, which may have only facts or facts *and* rules. In the last case, the rules are not going to be modified, as the BK is assumed as correct. In this way, every part of the model that is modifiable should be provided as the theory, and not as BK. The BK may change along with the stream. The set of facts is also called here as the knowledge base (KB). Finally, we have as input an element of the stream of examples, which may be a single example or a set of them. These are the examples that arrive together in a single time instant.

In case the model is modified within this single iteration of the lifelong process, both the updated ProPPR model and the tree representing its logical part are made available to external use and to the next iteration of the process.

The high-level procedure has two major components: the first one ($putInstanceIntoLeaves$) classifies the instance within each ProPPR example as positive and negative, according to the current model, and includes them into the appropriate tree leaves. ProPPR inference engine is used throughout the whole process of this component. After processing all the examples, the algorithm calls the second component ($revise$) that is responsible for proposing possible modifications to the current model and decide whether or not they are going to be, in fact, implemented.

Algorithm 2 OSLR: Online Structure Learner by Revision Top-level procedure

Input: Current ProPPR model (thy); Current representation of the clauses in thy as a tree ($tree$); Background Knowledge (BK), composed of (1) a non-empty set of facts and (2) a set of rules (possibly empty); An element of the stream of examples (with one or more examples) (E).
Output: The (possibly modified) ProPPR model (thy); The (possibly modified) tree representation ($tree$)

```
1: if $tree$ is empty then
2:     $tree \leftarrow$ start a tree from $thy$
3: end if
4: for each ProPPR example $\mathbf{e} \in \mathbf{E}$ do
5:     for each instance $e \in \mathbf{e}$ do
6:         $tree \leftarrow putInstanceIntoLeaves(tree, e, BK)$
7:     end for
8: end for
9: $thy, tree \leftarrow revise(thy, tree, BK)$
10: make $thy$ and $tree$ available for external use
```

As usual with approaches based on theory revision, we need to find and mark *where* the model needs to be modified, and *how* to modify them. As our approach is developed towards lifelong online learning, an essential step is, also, to decide *when* the model should be updated. The function called at line 6 in Algorithm 2 is partially responsible for the *where* issue, and to help on that we designed a theory representation tree, explained at Sect. 4.1. The *how* and *when* issues are taken care of by the function called at line 9 in Algorithm 2. Next, we give more details on these components.

4.1 Supporting the selection of where to modify the current ProPPR model

Finding the correct places to change in a theory is quite a challenging and expensive task (Paes et al. 2017). To smoothly do this task, we designed a tree structure that compactly represents the current theory, aiming at efficiently traversing it to suggest possible modifications. To achieve that, the nodes of the tree are associated with the literals appearing in the logical part of the clauses. In addition, there are special leaf nodes to represent the failure points of the theory, labeled as *false*. Each leaf in this tree holds a (possibly empty) set of instances: in case a clause is successfully used to prove an instance, a *non-false* leaf associated with the last literal of the clause is going to hold such an instance, together with its class, positive or negative. Otherwise, if a branch in the tree representing clauses with a sharing prefix fails to prove an instance, such an instance is established at a false leaf. In this way, we are able to select the points in the theory responsible for proving or not an instance.

A single tree is designed to represent rules with the same target predicates in their heads, i.e., in case the theory has multiple target concepts, where each concept is represented by a different predicate, there is going to be a tree for each one of them. Also, note that the features introduced in ProPPR language do not take part in the tree, as they act as latent variables to only cope with the weights of the rule.

The root of the tree represents the head of the rule, while the other nodes represent literals in the body of the rules. The first level of the tree is the head of the rule, the second level contains literals appearing in the first position in the body of the rules, the third level includes the literals appearing in the second position in the body of rules, and so on. Each non-false leaf is annotated with an identifier of the rule that ends at it (we omit this from the example, for clarity). A false leaf appears as the child of an internal node n_i in the tree, and represents the failure of the literals that come after the literal associated with n_i in the clause. In case

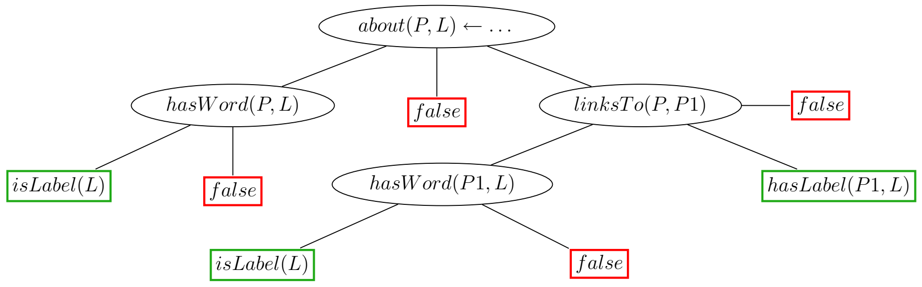


Fig. 2 Tree structure

n_i has more than one child, representing more than one clause with the same prefix up to n_i, the false leaf stands for the failure of all such clauses. Each path in the tree that does not end in a false leaf represents a single rule in the theory.

The tree that represents the example exhibited in Table 1 is shown in Fig. 2. As the leaves in the tree have a special role of representing either the end of the rule or a failure point, they are differentiated from the other nodes and graphically exhibited as squares. When a new instance arrives, we pass it through the tree, to decide where to put it. We start with the instance in the root, then we pass it through the nodes in the tree recursively as follows: for each node u in the children nodes of the current node v, we check if the (partial) rule from u to the root proves the example; if it does, we pass the example down to u. To do so, we look at the clauses that ProPPR inference engine returns as successfully proving the instance. If *none* of the children of v proves the instance, it is added to the set of instances of the false leaf, child of v. We repeat this process until the instance reaches the leaves of the tree. In case an instance is proved by more than one child node, it goes to all the nodes that prove it. This is the procedure included as a high-level command at line 6 of Algorithm 2.

The leaves of the tree indicate the points of the model that hold a potential to improve its score. In addition, the set of instances at each leaf are the ones that should be taken into account when modifying the model at that leaf. A *false* leaf in the tree contains instances that are not covered by the branch of the clauses ending at that leaf while the other leaves include the examples covered by its respective rule. In this way, positive examples in the *false* leaves and negative examples in the other ones are possibly misclassified by the current theory. We use this information to choose which part of the theory should be revised.

Note that when revising a purely logical model, we consider strictly the negative instances covered by the model and the positive instances not covered by it when selecting the revision points. Thus, a logical model is improved only if at least one proof for the non-covered positive instance emerges, or all the proofs for a negative instance disappear. A ProPPR model, on the other hand, benefits from both a disappearing of a path proof for a negative instance and an emerging of a successful proof for a positive instance, due to its random-walk based inference. Because of that, we annotate failure proof branches for a positive instance, even though it is covered by other clauses in the model, and, similarly, each successful proof path for a negative instance.

4.2 Revising a ProPPR model

The function called at line 9 of Algorithm 2, responsible for suggesting and possibly modifying the current ProPPR model and its accompanying tree, is presented as Algorithm 3. It starts by computing a heuristic value for each leaf, so that we may save time by selecting only the leaves with some potential to improve the model, and by choosing *first* the ones with more potential.

The heuristic is that the leaves with a larger number of potential improving instances are the ones with more potential to improve the model, if revised (Richards and Mooney 1995). If it is a *false* leaf, we compute the number of positive instances on it; when it is not a *false* leaf, we pick the number of negative instances. Next, we sort the leaves by the heuristic values, from the highest to the lowest. These processes are exhibited from line 2 to line 9 in Algorithm 3.

Then, for each selected leaf we proceed to suggesting modifications to it. A false leaf is an indicative of points of improvement for positive instances. By making the instances on it provable (or at least with more proofs), ProPPR inference engine may find more successful paths, which in turn increases the weight associated with the positive instances. This can be done by generalizing the theory. On the other hand, non-false leaves with negative instances are indicating that by removing such proofs we may improve the ProPPR model scoring. This can be achieved by specializing the theory.

The mechanism employed to suggest such modifications (lines 13, 15, 17, and 19 of Algorithm 3) are going to be explained at Sect. 4.2.1, when we present the revision operators developed in this work. Each proposed modification induces a new theory and a new accompanying tree. The instances associated with the leaf that has indicated the revision are also selected, as they are going to be used later to compute the new weights.

After suggesting each proper modification to the current leaf, they are scored (line 23 in Algorithm 3). To save time, we only re-learn the weights after choosing the modification to be implemented, following a structural EM-like procedure (Friedman 1998). Thus, in this step, we still use the default weights defined by the ProPPR weights' initialization parameter.

Next, the revision with the best score is chosen to be possibly implemented, at line 25. If it obeys the Hoeffding bounding restriction 4.2.2, then it is accepted to be the next model. Its new feature is generated (Sect. 4.2.3), and its new weights are computed, both using the set of instances associated with the revision (the same set associated to the leaf that has indicated the revision). These instances are discarded after the modification, while the rest of the instances in the tree remain unchanged in their respective leaves.

In case a modification is in fact implemented, the function accepts the new ProPPR model and its related tree. Then, it proceeds to the next leaf with the current ProPPR model and its accompanying tree. In the end, it may be the case that no one of the proposed revisions are chosen to be implemented and the function returns as output the same model received as input.

4.2.1 Revision operators: how the ProPPR model is modified

The leaves in the tree, together with their set of examples, point out where the theory should be updated. Here we present the mechanisms employed to suggest a modification to the theory using its accompanying tree, i.e., how the revision operators act.

Following the tree, this can be done by either adding nodes to the tree, or deleting nodes from it. Depending on the type of the leaf that we are looking at, by adding nodes we may either add a rule (generalizing the theory) or adding atoms to a rule (specializing it). Similarly, deleting nodes may lead to either deleting atoms from a rule (generalizing it) or deleting an entire rule from the theory (specializing it). Thus, by modifying the logical component of the model we expect to influence the probabilities of the predictions, since it may change the number of solutions in the graph and the number of paths from the starting node to a solution. Next, we explain how such operators are applied to the current model and the current tree.

Removing a node from the tree to generalize the theory (line 17 at Algorithm 3) This operator tries to gain more proofs to possibly uncovered positive instances by removing points that failed to cover them. We restrict the deletion of nodes to a single rule and a single atom aiming at preserving the upper part of the tree as most as possible, as it represents the old examples that have arrived at the stream, and also to keep the number of possible revisions at once as low as possible. Thus, we only allow a node to be removed if it obeys two conditions: (1) it is the last atom in the body of the rule and (2) it is the only sibling of the

Algorithm 3 Revise: Algorithm for suggesting modifications to a ProPPR model and possibly implementing them

Input: Current ProPPR model (thy); Current representation of the clauses in thy as a tree ($tree$); Background Knowledge (BK)
Output: The (possibly modified) ProPPR model (thy); The (possibly modified) tree representation ($tree$)

```
1: function revise($thy$, $tree$, $BK$)
2:     $leaves\_with\_potential \leftarrow \emptyset$
3:     for each $leaf \in tree$ do
4:         if $leaf$ has absorbed examples then
5:             $potential \leftarrow$ compute potential of $leaf$
6:             $leaves\_with\_potential \leftarrow leaves\_with\_potential \cup (leaf, potential)$
7:         end if
8:     end for
9:     $sorted\_leaves \leftarrow$ sort $leaves\_with\_potential$
10:    $revisions \leftarrow \emptyset$
11:    for each $leaf \in sorted\_leaves$ do
12:        if $leaf$ is false then
13:            $revisions \leftarrow revisions \cup \{(thy_1, tree_1)\}$ by adding nodes in $parent(leaf)$ and a clause in $thy$
14:        else
15:            $revisions \leftarrow revisions \cup \{(thy_2, tree_2)\}$ by adding nodes in $leaf$ and atoms in the corresponding clause in $thy$
16:            if $leaf$ is the only not false children of its parent then
17:                $revisions \leftarrow revisions \cup \{(thy_3, tree_3)$ by deleting $leaf$ from $tree$ and removing its corresponding atom from $thy$
18:            else
19:                $revisions \leftarrow revisions \cup \{(thy_4, tree_4)\}$ by deleting $leaf$ from $tree$ and removing its corresponding rule from $thy$
20:            end if
21:        end if
22:        for each $revision$ in $revisions$ do
23:            compute score for $revision$ using ProPPR and $BK$
24:        end for
25:        $best\_tree, best\_thy, best\_score, insts \leftarrow$ find the best theory in $revisions$ according to the best score
26:        if $best\_score$ obeys HoeffdingBound then
27:            find the new feature for $best\_thy$ when applicable
28:            compute new weights for $best\_thy$ using $insts$
29:            $thy, tree \leftarrow best\_thy, best\_tree$                ▷ accepts revision
30:            discard $insts$
31:        end if
32:    end for
33: return $thy, tree$
34: end function
```

false leaf that has pointed out the need of revision. The second condition is also responsible for not allowing redundant clauses in the theory (Duboc et al. 2017), as if a node has more than one child, deleting an atom from it would have the effect of making one of them to be subsumed by the other.

The operator works as follows: It starts from the not false leaf $leaf_i$, which matches both condition stated above, and proposes to delete it, making $parent(leaf_i)$ to become a leaf, and saving the instances to posteriorly score the revision. There is no need of maintaining $leaf_i$ in the tree.

Adding nodes to the tree to generalize the theory (line 13 at Algorithm 3) This operator also starts from a false leaf $f\text{-}leaf_i$ associated with a set of positive instances, which indicates that the theory may benefit from a generalization. Then, we add a new path $node_k \rightarrow \cdots \rightarrow node_{k+n}$ with n nodes in the tree, where $node_k$ is included as a child of the node $parent(f\text{-}leaf_i)$. At the same time, a new rule $label(root) \leftarrow \ldots label(parent(f\text{-}leaf_i)), label(node_k), \ldots, label(node_{k+n})$ is added to the theory.

To define the set of candidate atoms to compose the new rule, we follow a procedure similar to the generation of the Bottom Clause (Muggleton 1995). Thus, we transform the knowledge base (KB) in a graph GKB such that the nodes in this graph are the constants acting as terms of some fact in the the KB. In this graph, there is an edge between two constants t_i and t_j if there is a fact $atom(t_1, \ldots, t_i, \ldots, t_j, \ldots, t_k) \in KB, i, j \geq 1$. Even in the presence of facts with arity greater than two, we still represent them in the graph, by creating edges between all pairs of terms in the fact. For a fact with arity one, we include a loop edge, linking the node to itself.

We define the Bottom Clause (BC) of depth 0 created from an instance $p(t_1, \ldots, t_n)$ as the set of facts that originated the edges connecting any nodes to $t_1, \ldots, t_n \in GKB$. A BC of depth 1 is gathered from the BC of depth 0 plus the set of facts that originated the edges connecting the nodes that are 1 edge away from the nodes $t_1, \ldots, t_n$, i.e., the nodes that can be reached by going from a node $t_i \in \{t_1, \ldots, t_n\}$ to another node going through only 1 edge. For a BC of depth n, we include the edges from nodes that are n edges away from the example's terms.

To create the BC, we choose at random a ProPPR example **e** that is associated with the leaf indicating the revision. Then, we gather the positive instances belonging to **e** that are also associated with that leaf. This yields a set of instances of size m, where $m \geq |\mathbf{e}|$, where $|\mathbf{e}|$ is the total number of instances in **e**. Next, we compute each $BC_1, \ldots, BC_m$ of depth i, for each one of those positive instances. Then, we produce the final BC by putting them all together, i.e., the final Bottom Clause BC is $BC = BC_1 \cup \cdots \cup BC_m$. Finally, we replace the constants by variables.

Given the BC, there are two possible ways of including new atoms in a clause (and, hence, new nodes in the tree): the first one includes an atom at each time, choosing the one that better improves the score of the theory. We keep adding predicates, in a hill-climbing manner, until either the score can no longer be improved, after a predefined number of tries, or there are no more literals left to be added. The second way of creating a new rule is to find a path between the variables of the example, following the classical relational path-finding algorithm (Richards and Mooney 1992). We allow both algorithms to be executed concurrently and keep the modification that better improves the theory.

Adding nodes to the tree to specialize the theory (line 15 at Algorithm 3) This operator follows closely the one that creates a new rule, differing on where it starts in the tree and how the modification is implemented in the theory. The goal is to specialize a rule by adding atoms on it, aiming at excluding the proofs of the negative instances. It starts from a non-false leaf $nf\text{-}leaf_i$ that is associated with the proof of negative instances. Next, the candidate atoms are generated using the same Bottom Clause procedure described before, considering as starting point the positive instances that are in $nf\text{-}leaf_i$. In this way, we are trying to stop proving negative instances, while still proving those positive instances. After finding the set of literals $literal_1, \ldots, literal_n$ that better improves the theory, using either the hill-climbing or the relational path-finding procedure, they are appended to the rule that used to end at $label(nf\text{-}leaf_i)$. Thus, the node created from $literal_i$ becomes the child of $nf\text{-}leaf_i$, followed by the path $literal_{i+1} \rightarrow \cdots \rightarrow literal_n$.

Removing a node from the tree to specialize the theory (line 19 at Algorithm 3) To remove a rule from the theory and then eliminating proof paths in an attempt to avoid proofs of negative instances, we started from a non-false leaf $nf\text{-}leaf_i$ that has negative instances associated with. Here, we also try to preserve the upper part of the tree as much as possible, to possibly maintain a trace of the instances that are not in the tree anymore. Eventually, the new examples may continue to remove those old parts, but this is going to be showed along the time. Thus, we delete the $nf\text{-}leaf_i$ from $parent(nf\text{-}leaf_i)$, requiring that $parent(nf\text{-}leaf_i)$ has other children, so that the branch is not all removed at once. This has the effect of removing the rule that ends at $\ldots, \ldots, label(parent(nf\text{-}leaf_i)), label(nf\text{-}leaf_i)$ from the theory.

4.2.2 Hoeffding's bound: choosing when to implement a modification in the theory

After choosing the operator that most benefits the model, we have to decide if it is going to be in fact implemented, or if it is better to keep the model as it is. To support this decision, we use the Hoeffding's bound (Hoeffding 1963) to decide whether or not to implement the revision in the theory. Hoeffding's bound states that for a random variable r whose range size is R, and a set of n independent observations of this random variable, with empirical mean $\overline{r}$, $Pr(\mu_r \geq \overline{r} - \epsilon) = 1 - \delta$, where μ_r is the true mean of the random variable r, δ is a chosen parameter and ϵ is given by Equation 1:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \tag{1}$$

We use the difference between the evaluation of the two theories, considering the set of instances associated with the modified leaf, as the random variable. Consider a bounded metric M and a set of examples E containing n examples. Let T be the current theory and T' be the new theory proposed by revising T based on E. Let $\Delta M = M(T', E) - M(T, E)$ be the empirical difference between the performance of the new theory over the current one, given the examples and the metric.

Due to the Hoeffding's bound, we have, with probability $1 - \delta$, that $\Delta M_r \geq \Delta M - \epsilon$, where ΔM_r is the real mean of the random variable and ϵ is given by Eq. 1. Thus, if $\Delta M > \epsilon$ we have that $\Delta M_r > 0$, which means that, with probability $1 - \delta$, the new theory represents an improvement over the current one.

Summarizing, if the difference between the score of the new theory and the current one exceeds the threshold ϵ established by the Hoeffding's bound, given the number of examples, we assume that the proposed revised theory is better than the current one and such a revision is implemented at the current theory. Otherwise, we discard the suggested revision and follow to the next iteration of the loop. If the revision is accepted, the examples associated with the leaf used to propose the modification are discarded.

The random variable used in the Hoeffding bound statistics is the metric computed over the revision using a sample of the examples. The Hoeffding bound, as depicted in Eq. 1, depends directly on this sample size and assumes that it is computed over independent observations. In case we were dealing with identically and independently distributed (i.i.d) problems, we could compute the sample size of our random variable from the size of the examples. However, this is usually not the case of relational datasets (Jensen and Neville 2002). Thus, simply using the number of examples in the changed leaf as the size of the sample would overestimate it.

To alleviate this problem we adopted a subgraph sampling technique based on the Bottom Clause, inspired by Jensen and Neville (2003), where a form of conservative subgraph sampling reduced the bias caused by linkage and autocorrelation among sub-samples. With that, we aim at computing the number of *independent sets of examples*. To compose such sets, and consequently, the effective sample size, we use the subgraph sampling method, separating the instances according to their connected constants. While there is a connection through a constant between two instances, we consider them in the same set of dependent instances.

In this way, we consider that two ProPPR examples are dependent on each other if they have a node (a constant) in common in their respective Bottom Clauses, given a defined depth. Then, we use as the number of examples in the Hoeffding's bound equation this number of estimated independent examples. By definition, all the instances within a ProPPR example are related, since all of them have, at least, a constant in common. Thus, each ProPPR example counts as one independent example, even if such an example has more than one instance.

4.2.3 Feature generation

When changing a rule in the ProPPR model, it is necessary to re-define its feature. We propose an heuristic to create such a feature whenever we propose a revision on a rule: when a new rule is created, it has no feature, so a feature must be generated for it. When a rule has literals deleted from it or added to it, its feature may no longer makes sense, thus, the feature is discarded and a new one is created for it.

We can see the invention of new features as the problem of choosing the subset of the variables in a rule that better guides the PageRank-Nibble in the graph. Here, we propose to use the Jaccard index as the heuristic to decide which variables are better to be in this subset. To do that, first we get all possible θ-substitutions in the proof path of the examples used to propose the revision. Then, for each variable in the clause, we create two sets: the set of constants that substitute this variable and appear in the positive examples (named as $ConstantsInPositive$) and the ones that appear in the negative examples (named as $ConstantsInNegative$). Given these two sets, we calculate the value of the heuristic H using Eq. 2:

$$H = \frac{|ConstantsInPositive \cap ConstantsInNegative|}{|ConstantsInPositive \cup ConstantsInNegative|} \tag{2}$$

We compute the Jaccard index for each variable in the rule, then we sort the variable by its value, from the highest to lowest and find the biggest difference d between the heuristic of one variable to the next. Finally, we split this list of variables in d, getting two sets: the first one whose value of the heuristic is higher and the other where the value is lower. The first one becomes the variables of the invented feature.

Roughly speaking, this heuristic states that as higher the portion of the terms that appear in both positive and negative solutions the higher H will be, thus making the variable better to be considered in the features. While this may sound counter-intuitive, leveraging the intersection has the effect of reducing the weights of the rule during the ProPPR parameter learning, which in turn avoids the paths leading to the proof of both positive and negative instances. This is because the features give ProPPR more flexibility to define the probability distribution over the proofs since it can tune the weight of a path going through an edge given the rule (or fact) that was used to create such an edge.

After creating the new features to the implemented revision, we finally call the ProPPR parameter learning routine to learn the weights, given the examples used in the revision. This means that we train all the features related to this set of examples, including the new features and possibly some of the existing ones.

5 Experiments

In this section, we present the experiments that we have performed to evaluate the method proposed in this work. We have performed two types of experiments: the first one is a simulation of an online environment; the second one is a classical batch cross-validation approach. Next, we describe both of them.

5.1 Simulating the online environment

In these experiments, we have compared our proposal against RDN-Boost, an approach that learns Relational Dependency Networks proposed by Khot et al. (2015). We considered two datasets that they made available, namely, the Cora (Poon and Domingos 2007) and the UWCSE (Andersen et al. 2006) datasets. In the following, we describe the experimental methodology, the datasets, and the obtained results.

Additionally, we have compared our proposal against the HTilde system (Lopes and Zaverucha 2009; Menezes 2011) in the Same Paper relation from another version of the Cora dataset.

5.1.1 Methodology

We simulate the online incoming of examples by splitting the amount of examples in the dataset into iterations and by passing an iteration at a time to the learning algorithms.

Formally, given a Knowledge Base (KB) and a set of examples E, we split E into n iterations $I_1 \dots I_n$ of, approximately, the same size. Then we perform the evaluation similar to the Prequential approach (Dawid 1984). We start with an empty model and test it on the first iteration, then we pass this iteration to train the algorithm, it updates its model and then the model is tested on the next iteration. We repeat this process until no more iterations are available. In this way, every example from E will be used to evaluate the model before the model is able to train on it. In our experiments, we assume that the KB is fixed during the iterations, for simplicity. This assumption can be easily dropped and new facts can be added to the KB at the beginning of each iteration.

To test our online algorithm, this method is straightforward. However, to compare against a batch-learning method, we need to do an adaptation. We transform the N iterations on N learning tasks of the batch algorithm. Then, for each learning task $i \in [0, N-1]$, we create a training set composed of the examples $T = \bigcup_{k=0}^{i} I_k$ and test it on the examples from I_{i+1}; we assume that the iteration I_0 is an empty set of examples. Thus, we have each

Table 3 Size of the datasets

Relation	Positives	Negatives	ProPPR
Same Author	488	66	88
Same Bib	30,971	21,952	1295
Same Title	661	714	209
Same Venue	2887	4,976	403
Same Paper	31,429	807,149	1295
Advised By	113	16,601	91

model evaluated given the same information as the online evaluation, but we have to retrain the batch algorithm from scratch for every single iteration.

5.1.2 Datasets

In this set of experiments, we used three datasets, two versions of the Cora and the UWCSE as described below.

Cora is a dataset for citation matching (Poon and Domingos 2007). This dataset is composed of six background binary relations, represented by the following predicates: author, hasWordAuthor, hasWordTitle, hasWordVenue, title, and venue. Besides those, there are four target binary relations to be learned: sameAuthor, sameBib, sameTitle, sameVenue. We learned each one of the target relations separately, i.e., considering that each target relation is a separate dataset.

Cora–Same paper is another version of the Cora[2] used in Lopes and Zaverucha (2009) and Menezes (2011) to test the HTilde. This is a much larger dataset and has a single target relation, namely: Same Paper. Since HTilde requires a large number of examples, we only tested it on this dataset.

UWCSE dataset describes thirteen relations about the professors, students, and courses from the University of Washington (Richardson and Domingos 2006). The goal of this dataset is to predict the advisedBy relations between students and professors, given the other relations.

Before we can split our dataset into iterations, we have to convert the examples to the ProPPR's format. We do this by creating a query $p(a, X)$ for each target predicate p and constant a that appears in the first term of each example, then we group all the examples that have the same first term as the answers for their respective queries. Table 3 shows the number of examples for each relation, the first four lines are from the Cora dataset, which has 6540 facts in its BK; the next line is the average of examples per fold from the Cora Same Paper, which has approximately 231,500 facts and is divided into five folds of approximately the same size; and the last line is from the UWCSE, which has 12,615 facts. The last column shows the number of examples in the ProPPR's format. It is worth remembering that in the ProPPR's format, several instances are grouped into the same examples.

Since this dataset has many more negative than positive examples, we down sample the set of negative examples to be twice as much as the positives, following previous work that used this dataset (Khot et al. 2015). We only do not follow this procedure to tackle the Cora

[2] This dataset is available at http://dtai.cs.kuleuven.be/ACE/data/cora.zip.

Same Paper dataset, as we do not down sample the negatives examples to be consistent with Menezes (2011).

Finally, we split the relations into iterations. For the Cora datasets, we split them in such a way that the number of examples per iteration would lead to approximately 30 iterations. For the UWCSE, that has fewer examples than Cora, we place each example in ProPPR's format in one iteration, ending with 91 iterations.

All the experiments were run on PowerEdge R420 Intel Xeon 12 cores 2.20 GHz with 16 GB of RAM machines, making sure that the maximum number of jobs in a single machine were the number of cores. Each experiment was run 30 times and we report the average of those runs; at each time we shuffle the examples and for the UWCSE dataset we resample the negative examples. Since the Cora Same Paper is very large, and have already been divided into five folds, we run each online simulation from a different fold, ending up with 5 runs for this dataset.

For the ProPPR inference engine, we use the default values of ProPPR parameters $\alpha' = 0.1$, $\epsilon = 10^{-4}$ and a maximum depth of 20 vertices. On our algorithm we use the following parameters by default (unless explicitly specified otherwise): we pass all the examples of the iteration at once and try to propose modifications to all the leaves at the time of the revision, sorted by the potential heuristic; the depth of the bottom clause i is set to 1; to create the Bottom Clause, we use a single, randomly picked, positive example from the set of examples; as evaluation metric M, we try to optimize the area under the Precision–Recall curve; the Hoeffding's bound δ parameters is 10^{-3}, with a decay of 0.5 every time a revision is implemented; and we use the depth of 0 to define whether two examples are dependent on each other, so that we can generate a reasonable number of sampled subgraphs instead of assuming very few samples of estimated independent examples.

The RDN were run with the parameters made available by its authors along with the datasets. Since it has no parameters defined for the Cora Same Paper, we repeated the parameters of the Cora dataset.

5.1.3 Results

Figure 3 shows the evaluation of the area under the Precision–Recall curve for each relation, of the first Cora's dataset, over the iterations. A point of measure m and iteration i represents that m is the measure of the model, considering all the test examples until i, inclusive.

As shown by the charts, our approach outperforms the RDN on all the relations of the first Cora's dataset in both the area under the evaluation curve (AUC), which represents an overall view of the method on the task over the iterations, and at the final iteration (Final), which represents the performance of the model after using all the available data to learn. The values are considered statistically relevant by a two-tailed paired t test with $p < 0.05$, excepted by the final performance of the Same Author relation.

Furthermore, Fig. 4 shows the evaluation according to the area under the iteration curve for the relations of the first Cora dataset, considering different iteration sizes to split the set of examples. As can be seen, OSLR seems to benefit from iterations with fewer examples, indicating that it learns relatively better with fewer examples.

Figure 5 shows the results for the Cora Same Paper relation. We can see that HTilde obtained the best performance on this dataset, followed by the RDN and OSLR. However, until the third iteration, where fewer examples are available to train, we can see that OSLR has the best performance, contributing to our intuition that our approach needs less examples to achieve a reasonable result. The results are considered statistically relevant by a two-tailed

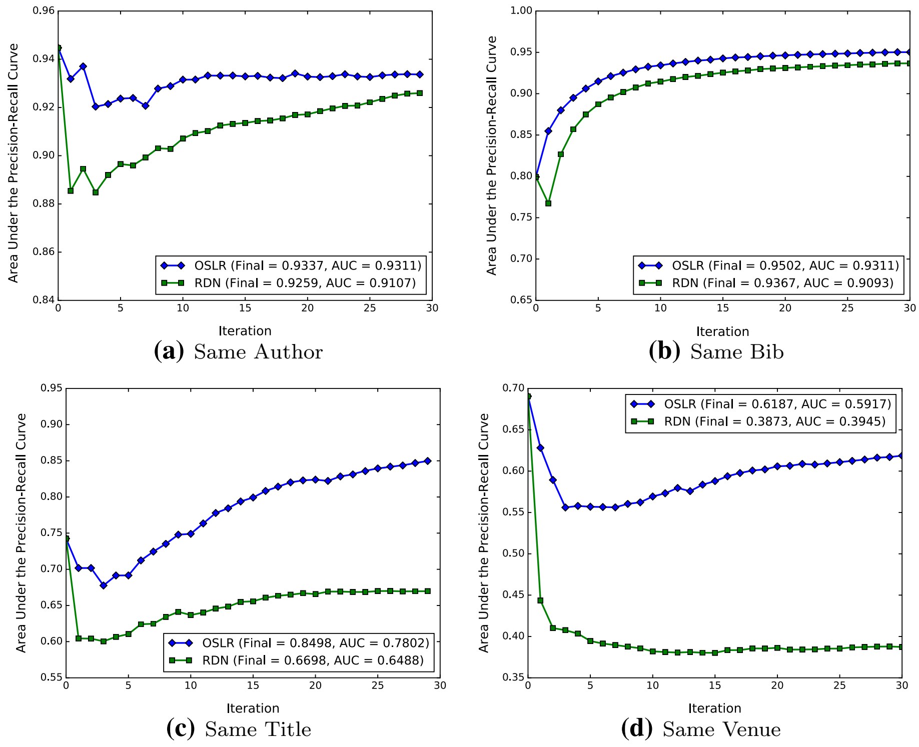


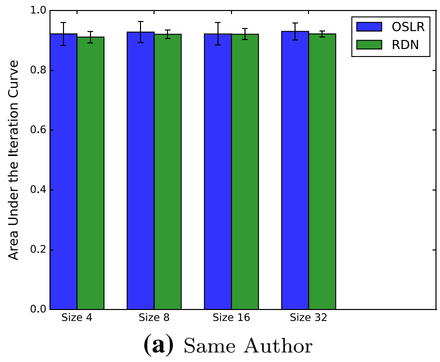
Fig. 3 The evaluation of the Cora dataset over the iterations

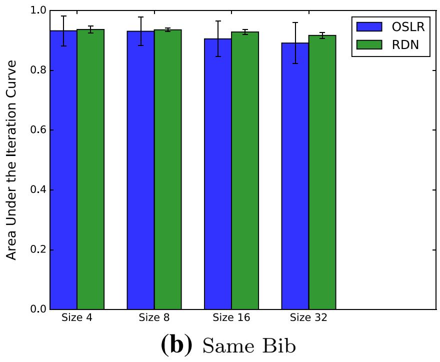
paired t test with $p < 0.05$, between any two pair of algorithms for the final evaluation and between HTilde and OSLR for the area under the evaluation curve.

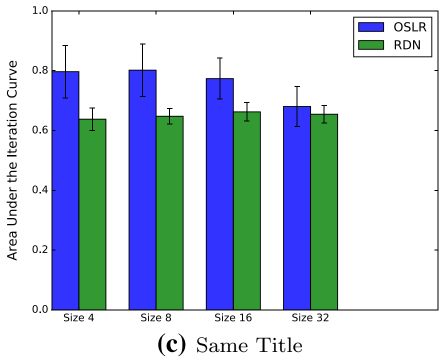
Finally, Fig. 6 shows the results obtained from the UWCSE dataset. In the left-hand Fig. 6a we can see that the RDN approach is slightly better than ours. In the right-hand Fig. 6b we present the methods learning from an initial theory. Instead of starting from an empty hypothesis, we started from an initial theory in the Background Knowledge.[3] The $RDN + I$ line is the RDN with the same initial theory and the Theory line is the performance of the theory itself (accordingly to ProPPR language) without any revision, and may be used as a baseline.

As we can see, by starting from an initial model, and allowing the revision system to modify it, we obtain slightly better results than the RDN without an initial theory. Furthermore, it is curious to notice that the RDN's performance decreases when the initial theory is put into its BK, while our performance increases. This might be caused by errors in the initial theory: since the RDN is not able to modify it, the errors are going to be taken to the final model. On the other hand, our approach is able to change the theory as needed, and keep improving its performance as new examples arrive, while the unmodified theory may get even worse with the new examples.

[3] This initial theory is manually written and it is available at http://alchemy.cs.washington.edu/. We modified it to conform with the expressiveness of our logic language.







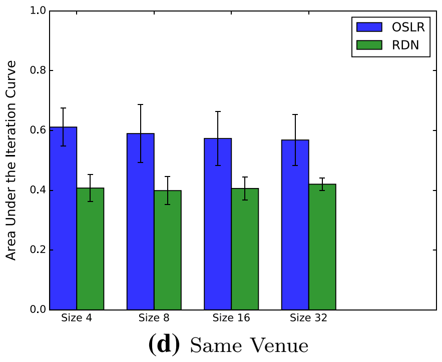
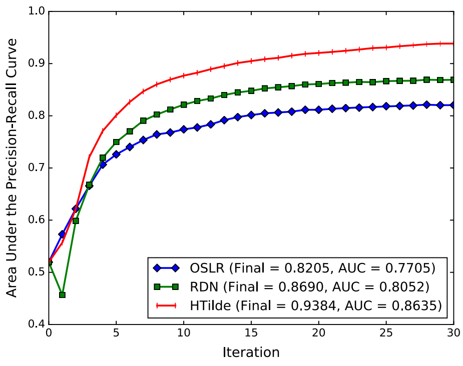


Fig. 4 The area under the iteration curve of the Cora dataset for different iteration sizes

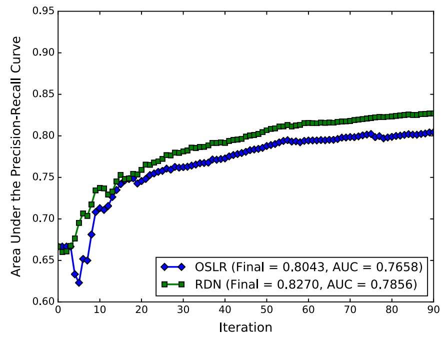
Fig. 5 The evaluation of the Cora Same Paper dataset over the iterations



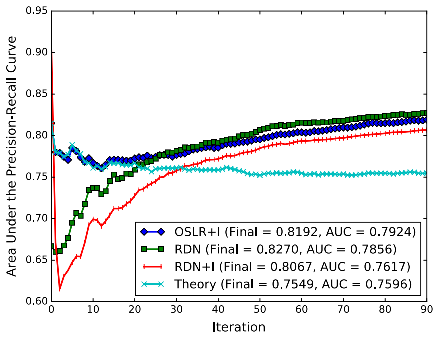
It is worth pointing out that there is no statistical significance among the results, based on a two-tailed paired t test with $p < 0.05$, in any of the comparisons between our approach and the RDN one in the UWCSE dataset.

Table 4 shows the average run time of each system on each relation. The RDN column is the average runtime of all the experiments, i.e. the N task per relation, where N is the number of iterations of the relation. The RDN Single column is the runtime of the last iteration, i.e. the runtime where the RDN sees the examples altogether.

The worst case for OSLR was in the Same Venue relation, where it takes, approximately, 2 times longer than RDN. This is the most difficult concept to learn and it seems it does



(a) UWCSE - Advised By



(b) UWCSE - Advised By with Initial Theory

Fig. 6 The evaluation of the UWCSE dataset over the iterations

Table 4 Average run time for each learned relations

Relation	OSLR	RDN	RDN single	HTilde
Same Author	0 min 19 s: 483	6 min 17 s: 979	0 min 18 s: 558	–
Same Bib	29 min 0 s: 606	1 h 18 min 1 s: 383	4 min 54 s: 538	–
Same Title	3 min 9 s: 413	10 min 31 s: 380	0 min 28 s: 891	–
Same Venue	1 h 24 min 0 s: 26	46 min 11 s : 78	1 min 53 s: 755	–
Advised By	9 min 0 s: 972	23 min 41 s: 939	0 min 8 s: 988	–
Advised By Init	5 min 59 s: 588	38 min 11 s: 854	0 min 21 s: 363	–

not take advantage of the benefits provided by the Hoeffding bound to avoid implementing revisions. In a closer look, we observed that, while in the fastest learned relation (Same Author), OSLR avoids proposing revisions to about 90% of the revision points, in the Same Venue, only about 20% of the possible revision points were not used to compute possible revisions. However, the greatest benefit of our approach relies on the fact that our system is always ready to make a prediction at any time, and keeps improving with the arrival of new examples. RDN, on the other hand, would need to run again before considering the new information to make predictions, at the expense of not representing the new examples.

Since we had to run the HTilde on a virtual machine, due to compatibility issues, we will not report the learning time for the Same Paper experiment. However, it was in the order of a few days, the same for our system, while it takes RDN a few hours.

5.2 Batch environment

In the previous subsection we showed the HTilde on the Cora Same Paper dataset in the online environment. In this section, we evaluate HTilde against our method in a batch environment reproducing the experiments from Menezes (2011).

We use a 5 × 2 fold cross-validation on the Same Paper relation. Each of these five folds has a knowledge base of approximately 231,500 facts and about 838,500 examples, as shown in Table 3, equally distributed in the two nested folds. For each of the five folds, we perform two pieces of training, using a nested fold at a time as the training set and the other as the test, having a total of 10 trainings per method.

Table 5 Area under the Precision–Recall curve for the Same Paper relation

Configuration	Measure
OSLR	0.8005 ± 0.0355
HTilde	0.9694 ± 0.0559
RDN-Boost	0.9051 ± 0.0239

This base is highly unbalanced, with the negative classes representing about 96% of the examples, but, as we have sad before, we did not down sample the negative class here because we would like to reproduce the results previously published at Menezes (2011).

Table 5 summarizes the results of this experiment. These results are the average over the 10 runs of the 5×2 fold cross-validation and has a statistical significance of both ours and RDN methods against HTilde on a two-tailed paired t test with $p < 0.05$. As can be seen, the HTilde outperforms (with statistical significance) both our method and the RDN approach. However, HTilde needs a large number of examples, while our method seems to benefit from small amount of examples, as it obtains better results on early iterations, where only a few examples were used to train the model. We were not able to use HTilde on the UWCSE dataset, for instance. Another point that it is worth mentioning is that HTilde proposes logic theories, bigger than our method, with several long rules, while our method's theories are usually composed by a few short rules, hence it is easier to understand their meaning.

6 Conclusions

In this work, we have proposed a novel method to online learn the structure written in ProPPR language, relying on theory revision techniques. In addition, we also proposed a heuristic to create the ProPPR's features. We have shown that the online results of OSLR outperform the state-of-the-art RDN-Boost system on the Cora dataset, and it is competitive to the RDN-Boost in the UWCSE dataset.

The application of theory revision techniques online brings two important benefits to the learning process: (1) OSLR is capable of continually improving the model with the arrival of new examples, and (2) it can benefit from an existent initial theory, even if this theory is only partially correct.

Our experiments have shown that our method performs well on a simulated online environment, even at the beginning of the learning process, where only a few examples are used to train the model. However, it has not performed so well in a batch environment, since it is not designed to act in this kind of environment.

As future work we would like to extend our algorithm to use other inference engines such as ProbLog (Dries et al. 2015) or Markov Logic Networks (Richardson and Domingos 2006). Another interesting future step would be to extend the system's logic language to handle non-monotonic knowledge by including, for example, negation-as-failure.

Acknowledgements We would like to thank the Brazilian Research Agency CNPq for the financial support and the authors of RDN-Boost, ProPPR, and HTilde for making their code available to us.

References

Andersen, R., Chung, F., & Lang, K. (2006). Local graph partitioning using PageRank vectors. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, USA, FOCS'06* (pp. 475–486).

Andersen, R., Chung, F., & Lang, K. (2007). *Local partitioning for directed graphs using PageRank* (pp. 166–178). Berlin: Springer.

Bifet, A., Holmes, G., Kirkby, R., & Pfahringer, B. (2010). Moa: Massive online analysis. *Machine Learning*, *11*, 1601–1604.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, *101*(1–2), 285–297.

Blockeel, H., De Raedt, L., Jacobs, N., & Demoen, B. (1999). Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, *3*(1), 59–93.

Cardoso, P. M., & Zaverucha, G. (2006). Comparative evaluation of approaches to scale up ILP. In *16th International Conference on Inductive Logic Programming (ILP 2006)* (pp. 37–39). Corunna: UDC Press.

Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T., Nigam, K., et al. (2000). Learning to construct knowledge bases from the world wide web. *Artificial Intelligence*, *118*(1–2), 69–113.

Dawid, A. P. (1984). Present position and potential developments: Some personal views: Statistical theory: The prequential approach. *Journal of the Royal Statistical Society Series A (General)*, *147*(2), 278–292.

De Raedt, L. (2008). *Logical and relational learning* (1st ed.). Berlin: Springer.

De Raedt, L., Kimmig, A., & Toivonen, H. (2007). Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th international joint conference on artifical intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, IJCAI'07* (pp. 2468–2473).

De Raedt, L., Kersting, K., Natarajan, S., & Poole, D. (2016). Statistical relational artificial intelligence: Logic, probability, and computation. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *10*(2), 1–189.

Domingos, P., & Hulten, G. (2000). Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD international conference on knowledge discovery and data mining, ACM, New York, NY, USA, KDD'00* (pp. 71–80)

Dries, A., & De Raedt, L. (2010). *Towards clausal discovery for stream mining* (pp. 9–16). Berlin: Springer.

Dries, A., Kimmig, A., Meert, W., Renkens, J., den Broeck, G.V., Vlasselaer, J., & De Raedt, L. (2015). Problog2: Probabilistic logic programming. In *Machine learning and knowledge discovery in databases—European conference, ECML PKDD 2015, proceedings, Part III, LCNS* (Vol 9286, pp. 312–315). New York: Springer.

Duboc, A. L., Paes, A., & Zaverucha, G. (2009). Using the bottom clause and modes declarations on FOL theory revision from examples. *Machine Learning*, *76*(1), 73–107.

Duboc, A. L., Paes, A., & Zaverucha, G. (2017). On the formal characterization of the forte_mbc theory revision operators. *J Log Comput*, *27*(8), 2551–2580.

Friedman, N. (1998). The Bayesian structural EM algorithm. In *UAI'98: Proceedings of the fourteenth conference on uncertainty in artificial intelligence* (pp 129–138). Morgan Kaufmann, Burlington

Gama, J. A., & Kosina, P. (2011). Learning decision rules from data streams. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence—AAAI Press, IJCAI'11* (Vol. 2, pp. 1255–1260).

Getoor, L., & Diehl, C. P. (2005). Link mining: A survey. *ACM SIGKDD Explorations Newsletter*, *7*(2), 3–12.

Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, *58*(301), 13–30.

Hulten, G., Abe, Y., & Domingos, P. (2003). Mining massive relational database. In *Proceedings of the IJCAI-2003 workshop on learning statistical models from relational Data* (pp. 53–60).

Jensen, D. D., & Neville, J. (2002). Linkage and autocorrelation cause feature selection bias in relational learning. In *Machine learning, proceedings of the nineteenth international conference (ICML 2002)* (pp. 259–266). Burlington: Morgan Kaufmann.

Jensen, D. D., & Neville, J. (2003). Autocorrelation and linkage cause bias in evaluation of relational learners. In *Inductive logic programming, 12th international conference, ILP. Revised papers*, Lecture Notes in Computer Science (vol. 2583, pp. 101–116), Berlin: Springer.

Khot, T., Natarajan, S., Kersting, K., & Shavlik, J. (2015). Gradient-based boosting for statistical relational learning: The Markov logic network and missing data cases. *Machine Learning*, *100*(1), 75–100.

Lopes, C., & Zaverucha, G. (2009). HTilde: Scaling up relational decision trees for very large databases. In *Proceedings of the 2009 ACM symposium on applied computing, ACM, New York, NY, USA, SAC'09* (pp. 1475–1479).

Menezes, G. (2011). HTilde-RT: Um algoritmo de aprendizado de árvores de regressão de lógica de primeira ordem para fluxos de dados relacionais. Master's thesis, PESC, COPPE, Federal University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil.

Mitchell, T. M. (1997). *Machine learning. McGraw Hill series in computer science*. New York: McGraw-Hill.

Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing*, *13*(3), 245–286.

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, *19*(20), 629–679.

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. Cambridge: MIT Press.

Natarajan, S., Khot, T., Kersting, K., Gutmann, B., & Shavlik, J. (2012). Gradient-based boosting for statistical relational learning: The relational dependency network case. *Machine Learning*, *86*(1), 25–56.

Paes, A., Revoredo, K., Zaverucha, G., Costa, V. S. (2005). Probabilistic first-order theory revision from examples. In *Inductive Logic Programming, 15th International Conference, ILP 2005, Bonn, Germany, August 10–13, 2005, Proceedings* (pp. 295–311).

Paes, A., Zaverucha, G., & Costa, V. S. (2017). On the use of stochastic local search techniques to revise first-order logic theories from examples. *Machine Learning*, *106*(2), 197–241.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab.

Poon, H., & Domingos, P. (2007). Joint inference in information extraction. In *Proceedings of the 22Nd national conference on artificial intelligence*, AAAI Press, AAAI'07 (vol. 1, pp. 913–918).

Pujara, J., London, B., Getoor, L., & Cohen, W. (2015). Online inference for knowledge graph construction. In *Workshop on statistical relational AI*.

Richards, B. L., & Mooney, R. J. (1992). Learning relations by pathfinding. In *Proceedings of the tenth national conference on artificial intelligence (AAAI-92), San Jose, CA* (pp. 50–55).

Richards, B. L., & Mooney, R. J. (1995). Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, *19*(2), 95–131.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine Learning*, *62*(1), 107–136.

Shapiro, E. Y. (1983). *Algorithmic program debugging*. Cambridge: MIT Press.

Srinivasan, A., & Bain, M. (2017). An empirical study of on-line models for relational data streams. *Machine Learning*, *106*(2), 243–276.

Tong, H., Faloutsos, C., & Pan, J. Y. (2006). Fast random walk with restart and its applications. In *Proceedings of the sixth international conference on data mining, IEEE Computer Society, Washington, DC, USA, ICDM'06* (pp. 613–622).

Tsunoyama, K., Amini, A., Sternberg, M., & Muggleton, S. (2008). Scaffold hopping in drug discovery using inductive logic programming. *Journal of Chemical Information and Modeling*, *48*(5), 949–957.

Wang, W. Y., Mazaitis, K., Lao, N., & Cohen, W. W. (2015). Efficient inference and learning in a large knowledge base. *Machine Learning*, *100*(1), 101–126.

Warren, D. H. D., Pereira, L. M., & Pereira, F. (1977). Prolog—The language and its implementation compared with Lisp. *SIGPLAN Not*, *12*(8), 109–115.

Wrobel, S. (2013). *Concept formation and knowledge revision*. New York: Springer.